The Thesis Committee for Anjana Subramanian
Certifies that this is the approved version of the following Thesis:

Advancing Value Prediction

APPROVED BY
SUPERVISING COMMITTEE:

Calvin Lin, Supervisor

Erez Mattan

Advancing Value Prediction

by

Anjana Subramanian

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2019

Dedicated to my beloved family.

Acknowledgements

Firstly, I'd like to thank my advisor, Prof. Calvin Lin for his guidance and encouragement. He has been influential in helping me discover my interests for computer architecture research. I'm thankful to him for this truly enriching learning experience.

I'd like to thank Prof. Erez Mattan who readily agreed to be the reader for my thesis despite his busy schedule.

I'd also like to thank Akanksha Jain, a postdoctoral researcher in Prof. Lin's group. From helping me submit jobs on the cluster to discussing advanced research ideas, she has played a key role in this journey.

I'd like to thank Pawan Joshi, a fellow researcher, for the numerous discussions we have had on value prediction. I'd also like to acknowledge the larger research group that works for Prof. Lin. I have learnt tremendously from their work presented in weekly meetings.

Finally, I am thankful to Amma, Pappa, Karthik and Aayu. If not for their love, I would not be here.

Abstract

Advancing Value Prediction

Anjana Subramanian, M.S.E

The University of Texas at Austin, 2019

Supervisor: Calvin Lin

Read after write dependencies form a key bottleneck in single thread performance. Value prediction [9][10][18] is a speculative technique that overcomes these dependencies by predicting results of instruction execution, thereby preventing dependent instructions from stalling. Usually, the penalties for value mispredictions are extremely high. As a result, value predictors have evolved to prioritize accuracy over coverage. To improve upon the state-of-the-art, our goals are: (i) to develop more powerful prediction mechanisms that have a better accuracy-coverage tradeoff (ii) to maximize performance gains obtained from correct predictions. We present two independent pieces of work that address each of these.

To achieve the first goal, we design a Heterogeneous Context-based Value Predictor (HCVP) that combines the use of branch history with value history to represent program context information. We demonstrate that this combination provides better predictability than using either of them individually and that it allows for the use of relatively short value history lengths that provide more coverage than very long ones. HCVP does not maintain speculative value histories as it more tolerant to the update

problem that occurs when back to back instances of the same instruction are predicted. Our predictor performs better than the state-of-the-art value predictors (E VTAGE and DFCM++) to achieve a 29% speedup over a baseline with no value prediction. When combined with the E Stride predictor, it achieves a speedup of 46%, which is 9% higher than that achieved by E VTAGE E Stride (EVES), the winner of the First Championship Value Prediction.

To achieve the second goal, we exploit the fact that some instructions are more performance critical than others. We categorize instructions by various parameters to find one or more classes of instructions that provide high performance benefits for correct predictions. We find that loads, address producing instructions, and high fanout instructions are extremely beneficial for value prediction.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

As the benefits of parallel computing are limited by the parallelizability of software [13] and the migration of the software industry towards parallel application development has been slow, the need for improving single thread performance continues. To that end, numerous microarchitectural techniques such as pipelining, out-of-order execution, branch prediction and superscalar architectures have been explored. The primary motivation behind these techniques is to improve performance by increasing instruction-level parallelism (ILP). Despite enhancing hardware designs to achieve ILP, read after write (RAW) dependencies have to be strictly enforced in most cases. These dependencies occur when source values of instructions are dependent on the execution results of previous instructions. Under such circumstances, instructions stall and execution is serialized. This creates a severe bottleneck in processor performance.

Value prediction [9] [10] [18] is a speculative technique that can potentially overcome RAW dependencies by predicting results of instructions. These predicted results can be used as source values for dependent instructions, thereby preventing them from stalling. This can shorten critical paths in program execution and can lead to better performance of sequential code. To enforce correctness, a mechanism to recover from incorrect predictions is required.

Perfect value prediction performed on a set of 135 traces from the CVP simulation infrastructure [7] achieves a speedup of 2.5 times over a baseline with no value prediction.

Although value prediction was first explored in the 1990s [9] [10] [18], it has not been adopted widely. Key concerns have been with respect to accuracy of predictions and complexity of the pipeline required to support value prediction. However, recent work [1]

[3] introduces predictor designs with extremely high accuracy (~99%), allowing for the use of simple misprediction recovery mechanisms that impose high penalties. Further, recent work also demonstrates that value prediction can pave the way for an out-of-order engine that is less aggressive, thereby reducing pipeline complexity [6]. These developments have generated renewed interest in pursuing value prediction as a potential technique to improve hardware performance.

1.1 GOALS

Value prediction is generally considered to be a difficult problem because the average performance gain per correct prediction tends to be low while misprediction penalties tend to be extremely high (~5-50 cycles) [1]. The choice of misprediction recovery mechanism is usually a tradeoff between hardware complexity and the cost (latency) of misprediction. To simplify complexity, a pipeline flush is preferred where all instructions following the mispredicted one are re-fetched and re-executed. To deal with these challenges, predictor designs have focused on achieving extremely high accuracies (~99%) [1] [3]. In this process, we believe that predictors have resorted to conservative mechanisms that compromise on coverage to achieve the desired accuracy.

For instance, the recent DFCM++ [8] predictor tries to capture program context using PC localized value histories. The predictor learns correlated value patterns and predicts that the same pattern will occur again when a similar value history is observed. However, it uses very long value history lengths of 32 and 64. We demonstrate that the ability to identify patterns and predict the values of a larger number of instructions (i.e., coverage) tends to be low at these history lengths, owing to the large training times involved. However, they are preferred since they provide improved accuracy and hence better speedup.

Another instance is the VTAGE predictor [5] that chooses to use global branch history over PC localized value history. Predicting back to back instances of the same instruction can lead to inaccuracies when value history is used. VTAGE avoids this problem by using global branch history instead. However, we demonstrate that branch history is less powerful than value history is in detecting patterns. In other words, the use of branch history trades off coverage for accuracy.

One final example is the use of confidence counters in predictor designs. These counters are trained based on the correctness of speculated values stored in the table. The stored values are used only when the counters saturate, indicating high confidence in prediction. While the counters are still training, coverage is lost. However, accuracy is prioritized and hence predictors are usually designed to take this hit in coverage.

In summary, the high bar for accuracy has caused the evolution of value predictor design to take a more defensive approach. Our goal is to improve the aggressiveness of value predictors through two techniques:

- Improve predictability so that sufficient accuracy can be achieved without hurting coverage
- Increase the gains per correct prediction so that the bar for accuracy is lowered

This thesis presents two independent pieces of work that targets each of these goals.

1.2 OUR SOLUTION

In our first piece of work, we design a predictor with improved prediction capabilities. We use a systematic approach to explore predictability limits when either branch history or value history is used to capture program context. We also explore predictability limits across context lengths. For a value to be predictable,

- it has to follow a context enough number of times to build prediction confidence
- it has to follow the context continuously without being interrupted by another value that follows the same context. (We use the term divergence to describe a situation where the same context is followed by different values at different times)

Based on these criteria, we establish that PC localized value histories of lengths 4 to 16 provide the best predictability (i.e., the best coverage and divergence handling capability). However, it does not translate into the best speedup as a result of insufficient accuracy.

We explore the use of branch history combined with value history and demonstrate that it achieves better accuracy and divergence handling capabilities without significant loss in coverage when compared to using value history alone. We also demonstrate that it has better tolerance to the update problem that arises when different instances of the same static instruction have to be predicted back to back. We introduce the Heterogeneous Context-based Value Predictor (HCVP) designed based on these observations and empirically compare against other state-of-the-art value predictors.

In our second piece of work, we identify one or more instruction classes that provide high average performance gain per correct prediction. To study available headroom, we categorize instructions using different parameters and apply perfect value prediction on each of them. We find that some instruction classes provide high performance benefit per correct prediction. Among them, a few provide sufficient coverage while a few others are so few in number that it is not worthwhile pursuing them exclusively.

1.3 CONTRIBUTIONS

This thesis makes the following contributions:

- We systematically evaluate the predictability of using either value history or branch history to represent program context information. While we identify that PC localized value history with global application provides the best predictability, we also illustrate how it falls short of its true potential due to inaccuracy and inability to deal with diverging value streams.
- We demonstrate that combining the use of branch history with value history improves its divergence handling capability and accuracy resulting in better predictability[1] and performance. It also allows for the aggressive use of shorter value history lengths that provide better coverage. We introduce the Heterogenous Context-based Value Predictor (HCVP) based on these principles.
- We experimentally demonstrate that value histories localized by both PC and branch history are more tolerant to the update problem as opposed to using value histories localized by PC alone. As a result, HCVP does not maintain any speculative value histories.
- We show that HCVP performs better than the current state-of-the art value predictors (E VTAGE and DFCM++), providing a speedup of 29% against a baseline with no value prediction. When combined with the E Stride predictor, it achieves a speedup of about 46% which is 9% higher than that of E VTAGE with E Stride (EVES), the winner of the First Championship Value Prediction (CVP-1).

[1] Some of the conclusions on improved predictability using heterogenous context information were arrived at independently by Joshi in his thesis [17] as well. While this thesis derives its conclusions from a systematic analysis of the gaps in performance between realistic and oracle divergence handling capabilities, Joshi's work derives them based on the combined benefits of using EVES and DFCM++ and based on a "variance" metric that roughly estimates divergence.

- In the second piece of work, we empirically find that loads, address producing instructions, and high fanout instructions are extremely beneficial for value prediction. While some of these ideas have been known before, we quantify tradeoffs in coverage and average benefit per correct prediction. Until now, high fanout instructions have not been specifically targeted for value prediction.

Chapter 2: Related Work

This chapter provides an overview of prior work in value prediction. Value prediction was introduced independently by Lispasti et al. [9] [18] and Gabbay and Mendelson [10]. It is based on the observation that programs exhibit value locality. In other words, results produced by instructions form a predictable pattern. For instance, an instruction can produce values that are constants or form a strided pattern. Value predictors have been designed to make predictions based on patterns observed in the past.

Broadly, value predictors can be classified into two categories – computational predictors and context-based predictors [14]. Computational predictors compute values by applying a function to previously seen values. Based on previous history, they learn when it may be suitable to apply one or more functions to predict a value. Context-based predictors memorize patterns. They learn values that follow a certain context and predict the same value when the context repeats. Some of the state-of-the-art predictors are hybrid predictors that combine a context-based predictor with a computational predictor [3] [8].

Most value predictors employ confidence mechanisms to achieve high prediction accuracy [3] [8]. This is usually done by maintaining confidence counters for each prediction stored in the predictor tables. These counters are trained based on whether the stored predictions match the actual execution results. The stored predictions are used only when the confidence counters saturate indicating high confidence in prediction.

Section 2.1 describes a few computational predictors while section 2.2 describes a few context-based ones. Section 2.3 describes the hybrid predictors.

2.1 COMPUTATIONAL PREDICTORS

2.1.1 Last Value Predictor (LVP)

This is the simplest kind of predictor [18]. For a given instruction PC, it predicts that the value will be same as the value seen during the previous instance of the instruction. In other words, it performs the identity operation on the last seen value. The predictor consists of a table that is indexed using the lower bits of the instruction address. Each entry contains a tag and a last value. The higher bits of the instruction address are compared against the tag. If a match is found, the corresponding last value is used as the prediction. After the instruction completes execution, the last value of the corresponding entry in the table is updated.

2.1.2 Stride Predictor

The stride predictor [10] table is similar to the table used by the Last Value Predictor, except that entries contain an additional field called stride. The stride value is added to the last seen value to obtain a prediction. Once an instruction completes execution, the stride field is updated by computing the difference between the current result and the last value observed for the instruction. The last value field is then updated to reflect the current result. This predictor is useful for predicting values of instructions that exhibit a strided pattern. Examples include loop iteration variables or indexes used for sequential array accesses.

2.1.3 2-Delta Stride Predictor

The 2-delta stride predictor [21] is a variation of the stride predictor described above. Each entry consists of two strides instead of one. The first stride is updated always whereas the second stride is updated only when there is no change in value for the first

stride. The second stride is used for prediction. The predictor is designed this way to reduce mispredictions for loop iteration variables when new instances of the loops begin. For every new instance of a loop, the baseline stride predictor will encounter two mispredictions – one for the first iteration and one for the second iteration. 2-Delta Stride Predictor will mispredict only for the first iteration.

2.1.4 gDiff Predictor

Unlike the previous predictors, the gDiff predictor [22] looks at global value history to make a prediction. It computes the differences between the result of an instruction and the results of the last n instructions. If any of the computed differences match a previously seen difference, then the matching difference along with the last seen value of the corresponding instruction in global history is used to make a prediction.

2.1.5 Value Estimator

The Value Estimator [8] tries to infer operations (like addition or subtraction) based on past history of source and destination values corresponding to instruction PCs. It applies the inferred operation on the source operands to compute the resultant value. In case the source operands are not ready, it applies the same mechanism to infer source values by looking at their producer instructions.

2.2 CONTEXT-BASED PREDICTORS

2.2.1 Finite Context Method (FCM) Predictor

FCM [19] is a two-level predictor that uses local value history to represent context information. The first level table is called as the Value History Table (VHT) and is indexed using the instruction address. Each entry in this table contains the last n values observed for the given instruction. In other words, they contain the local value history.

These values are hashed to index into a second level table called as the Value Prediction Table (VPT). The Value Prediction Table consists of the actual prediction. After the instruction executes and the result is known, the corresponding value history and prediction are updated.

2.2.2 Differential Finite Context Method (DFCM) Predictor

DFCM [15] is a variant of FCM that tracks differences between the values instead of the values themselves. An entry in the VHT consists of a local history of differences. A Last Value Table (LVT) is maintained to keep track of the last seen result for a given instruction address. An entry in the VPT Table contains a difference that needs to be added to the last seen value to obtain a prediction.

2.2.3 Value TAGE (VTAGE) Predictor

VTAGE [5] derives from the popular ITTAGE Branch predictor [12] and leverages similarities between branch target prediction and value prediction. It uses instruction address and global branch history to represent context information. The predictor consists of a base table and several tagged tables. The base table is indexed using the instruction address whereas the tagged tables are indexed using a hash of the instruction address and global branch history. Longer branch history lengths are used to index higher order tables. Each entry in the base table and tagged tables stores the last seen value for the given context and a confidence counter. The confidence counter is a saturating counter that is incremented for correct predictions and decremented for wrong ones. Predictions are made only when the confidence counter values are high. In addition to these, the entries in the tagged tables consist of a partial tag and a useful counter. The

partial tag is used to match higher order bits of the indexing value. The useful counter is used by the replacement policy when entries have to be evicted.

At prediction time, all the tables are searched in parallel. A matching component that uses the longest branch history is used for the prediction.

2.2.3 Differential Value TAGE (VTAGE) Predictor

The DVTAGE predictor [1] modifies the tables in the TAGE to store strides instead of values. It also augments it with a Last Value Table (LVT). It predicts values by adding the predicted stride with the last value for the given PC. It maintains a speculative window to keep track of last values of inflight instructions.

2.3 HYBRID PREDICTORS

2.3.1 Enhanced VTAGE Enhanced Stride (EVES) Predictor

EVES [3] is the winner of the First Championship Value Prediction [7] in all categories (i.e., 8kB, 32kB and unlimited). It is a hybrid predictor that combines enhanced versions of VTAGE and Stride. Some of the enhancements to VTAGE include the addition of partial tags to the base table and compression of data values. Other enhancements include careful confidence management and entry allocation based on expected average performance benefit of a correct prediction.

A key enhancement to the Stride Predictor includes the computation of the predicted value as a function of not only the last value and the predicted stride, but also of the number of inflight instructions for the given instruction address.

The VTAGE component is preferred over the Stride component when both of them provide confident predictions.

2.3.2 Differential Finite Context Method++ (DFCM++) Predictor

The DFCM++ Predictor [8] was the runner up in the unlimited size category of the first Championship Value Prediction [7]. It augments the DFCM predictor [15] with several enhancements. It maintains speculative histories of differences and speculative last seen values to prevent mispredictions due to the delay between making a prediction and knowing the actual result. It also maintains histories of multiple lengths and chooses between them dynamically. To keep track of instruction PCs whose values are frequently mispredicted, it maintains a PC Blacklister.

It combines this component with a Value Estimator and prioritizes the latter if it can make a confident prediction.

Chapter 3: Improved Context-Based Value Prediction

3.1 INTRODUCTION

Majority of the state-of-the-art value predictors are context-based [1] [3] [5] [8]. They learn values that follow a certain context and predict them when the same context repeats [14]. Context-based predictors differ in the information they use to represent program context. Broadly speaking, there have been two categories – ones that use branch history [1] [3] [5] and ones that use value history [8] [15] [19]. Like any other design choice in computer architecture, there are tradeoffs in using either one of these for prediction. For instance, value history-based predictors are more suited to capture any arbitrarily correlated value pattern while branch history-based predictors are generally limited to capturing constant or strided patterns. As a result, the former tends to have greater coverage than the latter. On the other hand, predicting using value history requires knowledge of recent values. Some of these values may not be available if the corresponding producer instructions have not completed execution. We call this as the update problem because it results from the predictor state not being updated in time to reflect correct value history. Branch history-based predictors don't suffer from the update problem [5] as they rely on recent branch history that is always available based on actual or predicted branch outcomes. A mispredicted branch is not a concern as a pipeline flush occurs causing instructions to be re-fetched. In other words, branch history-based predictors do not suffer from prediction inaccuracy due to non-availability of recent values.

In this chapter, we explore the predictability and performance impacts of using branch history or value history to represent context information. We explore different context lengths and quantify tradeoffs in accuracy, coverage and speedup. Our key

contribution is in demonstrating that a combination of branch history with value history works better than either one of them. It helps achieve the coverage benefits of using value history while being more tolerant to the update problem. Based on these observations, we introduce the Heterogenous Context-based Value Predictor (HCVP) that uses value history localized by both PC and branch history.

While this work focuses on improving context representation for better predictability and performance, it does not focus on the practicality of predictor implementation with respect to storage budget. We assume that we have infinite storage budget and evaluate against the state-of-the-art by selecting winning entries from the unlimited storage track of the First Championship Value Prediction (CVP-1) [7]. We consider limiting predictor table sizes as future work.

Our key contributions can be summarized as follows:

1. We explore the design space for context-based predictors to evaluate predictability limits. We demonstrate that the use of PC localized value history with global application provides the best predictability among known ways to represent context information in current value predictors.
2. We demonstrate that the use of value history for prediction loses out on significant coverage and accuracy due to divergences in value streams. There is further potential to improve performance if divergences can be dealt with more effectively.
3. We demonstrate that a combination of branch history and value history provides better divergence handling capabilities than using value history alone. It achieves improved accuracy with negligible drop in overall coverage, despite having to train over more context information.

4. We achieve peak performance using a combination of 16 values with global branch history. However, we show that a combination of 4 values with branch history is sufficient to match the performance of using long value histories (32 values).
5. We demonstrate that the combined use of value history and branch history is more tolerant to the update problem than value history alone is. As a result, speculative value histories need not be maintained.
6. We evaluate our predictor, HCVP, to show that it outperforms state-of-the-art context-based predictors, namely E VTAGE [3] and DFCM++ [8] by achieving a speedup of 29% over a baseline with no value prediction. Further, we combine it with the E Stride predictor to achieve a speedup of 46% which is 9% higher than that of E VTAGE E Stride (EVES) [3], the winner of CVP-1.

3.2 Predictability of Context Based Predictors

In this section, we explore the design space to compare different representations and lengths of context information to select our baseline. Our goal is to select a baseline that has the best predictability. We expect it to perform well in terms of capturing patterns (coverage) and making correct predictions based on them (accuracy). We rely on the speedup metric to reflect the combined effects of accuracy and coverage. To evaluate predictability efficiently, we ignore the update problem throughout Sections 3.2 and 3.3. We assume that an oracle update mechanism exists i.e., the correct value is known immediately after making a prediction and hence the predictor state is up to date at any given point of time. Eventually, we re-evaluate with a realistic update mechanism in Section 3.4.

3.2.1 Divergence

The generic structure of a context-based predictor is shown in Figure 1. It shows the prediction operation. An entry in the table consists of a prediction and a confidence counter and is selected based on the context. The prediction is made use of only if the confidence counter exceeds a threshold.

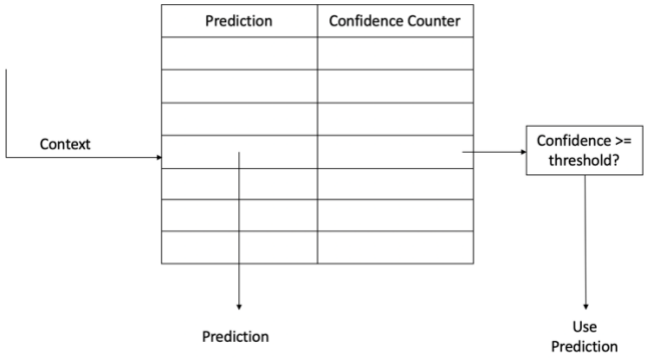

Figure 1: Generic Structure of a Context-based Predictor

Figure 2 explains the basic algorithm for predictor table update when the actual value is known. The confidence counter is incremented if the prediction stored in the table is correct. It is reset to zero if it is incorrect. The prediction is replaced with a new one if the counter is already at zero.

```
if (actual_value is consistent with prediction)
        confidence_counter = min (confidence_counter + 1, threshold)
else if (confidence_counter != 0)
        confidence_counter = 0
else
        update prediction to make it consistent with actual value
```

Figure 2: Update Mechanism for a Context-based Predictor

Based on this generic design, we see that the following two are requirements for good predictability:

Requirement 1: Contexts are followed by a value enough number of times (>= threshold) so that the confidence is high enough to make a prediction

Requirement 2: Contexts are followed by the same value all the time. In practice, the same context can be followed by different values at different times. We call this as "divergence".

Note that both these requirements depend on the quality of information used to represent the context. When program context is captured well, these requirements are satisfied to a large extent resulting in high predictability.

There is potential to improve context representation to deal with divergences. For instance, additional context information can be used along with existing information to split diverging streams into separate entries in the table so that they are no longer diverging. If we satisfy requirement 2 this way, requirement 1 establishes an upper bound on predictability. To quantify this upper bound, we introduce the concept of "oracle divergence handling".

A hypothetical predictor with oracle divergence handling capability remembers not one, but many predictions for each context. It can make a correct prediction as long as it has seen the correct value follow a given context enough number of times (>=threshold). It uses an oracle mechanism to select the appropriate prediction. By design, it operates at 100% accuracy. As a result, confidence counters are never reset to zero, they are only trained positively when values repeat.

We call the more realistic predictor described using Figure 1 and Figure 2 as the one with "practical divergence handling".

3.2.2 Design Space Exploration

We look at variants in predictor design based on how context information is represented. First, we classify them depending on whether branch history or value history is used. Further, we classify value history-based predictors based on whether they apply patterns learnt from a PC globally (across PCs) or locally (within the same PC). All of these predictor designs can further be classified based on whether they predict values or differential values (i.e., strides). We describe each one of these designs in more detail.

3.2.2.1 Value History: PC Localized Learning, PC Local Application

The context information used is PC, PC Local Value History. Predicted values are stored in the prediction table.

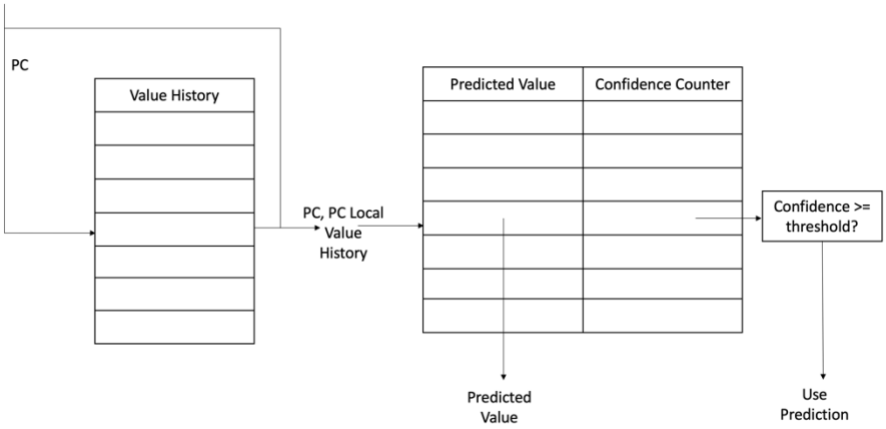

Figure 3: Value History: PC Localized Learning, PC Local Application

3.2.2.2 Differential Value History: PC Localized Learning, PC Local Application

The context information used is PC, PC Local Differential Value History. Predicted strides are stored in the prediction table.

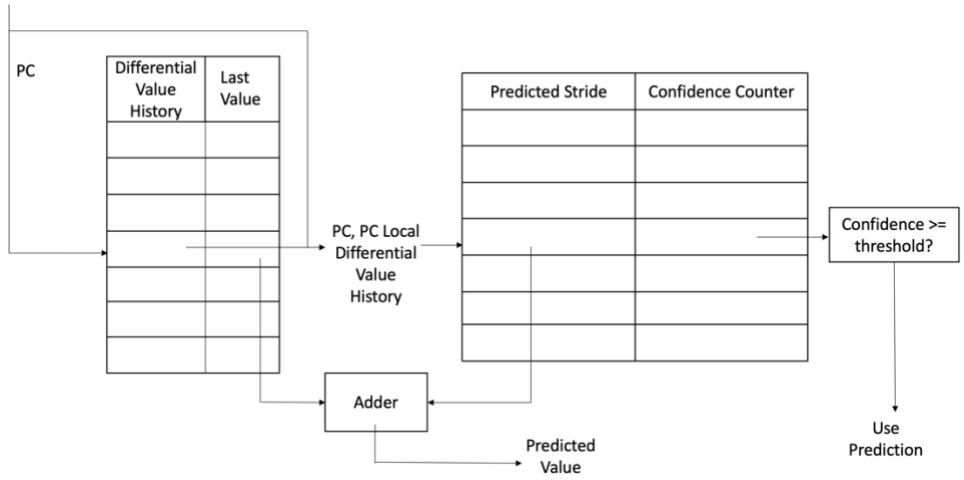

Figure 4: Differential Value History: PC Localized Learning, PC Local Application

3.2.2.3 Value History: PC Localized Learning, Global Application

The context information used is PC Local Value History. Predicted values are stored in the prediction table.

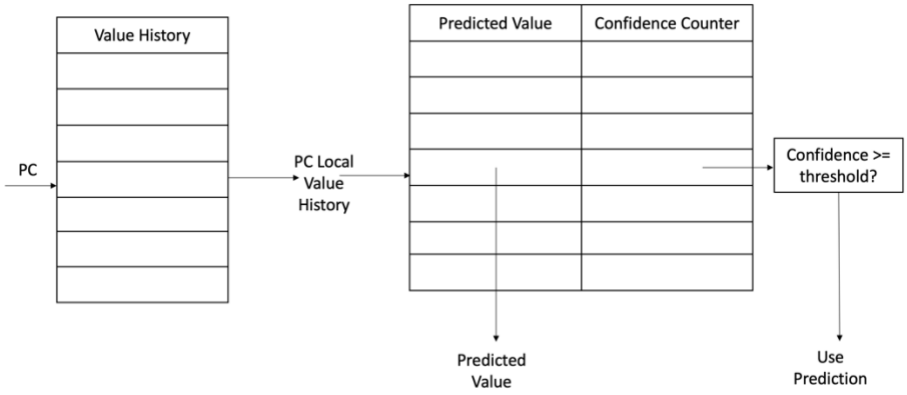

Figure 5: Value History: PC Localized Learning, Global Application

3.2.2.4 Differential Value History: PC Localized Learning, Global Application

The context information used is PC Local Differential Value History. Predicted strides are stored in the prediction table.

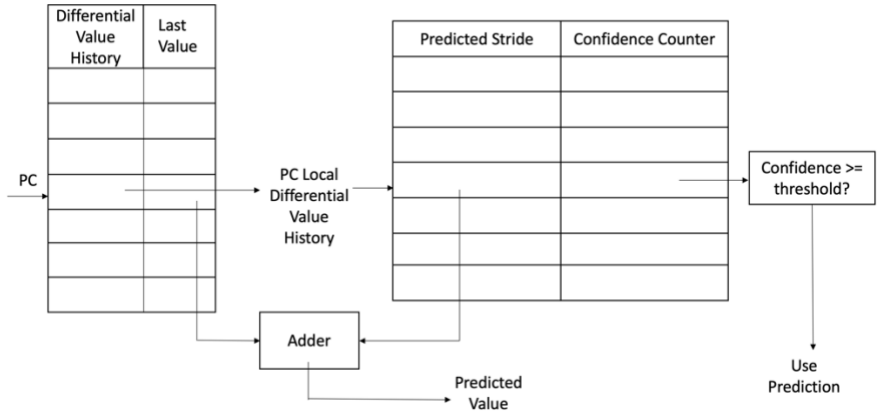

Figure 6: Differential Value History: PC Localized Learning, Global Application

3.2.2.5 Branch History: Value Prediction

The context information used is PC, Global Branch History. Predicted values are stored in the prediction table.

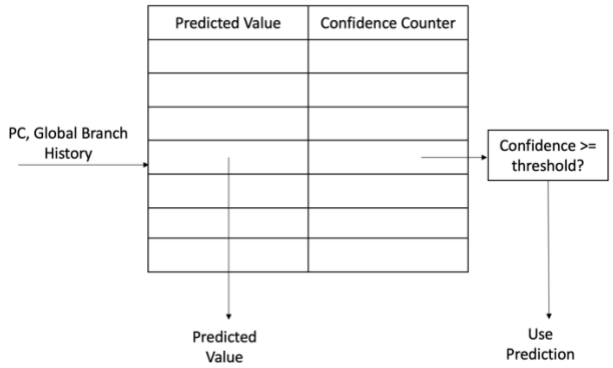

Figure 7: Branch History: Value Prediction

3.2.2.6 Branch History: Differential Value Prediction

The context information used is PC, Global Branch History. Predicted strides are stored in the prediction table.

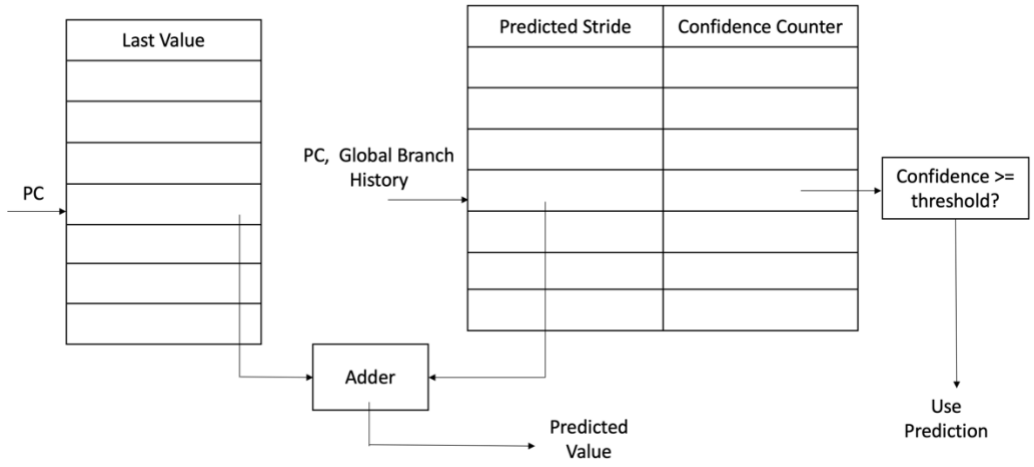

Figure 8: Branch History: Differential Value Prediction

3.2.2.6 Other Variants

We do not evaluate global value history-based predictors as the update problem is extremely severe in them. Since most of the recent values in the global history may be unknown at prediction time, the predictor tends to be very inaccurate when a realistic update policy is applied.

3.2.2.7 Differential Values (Strides) vs Values

First, we evaluate between using differential values (strides) and values. When using strides, the predictor keeps track of the last observed value for a given context and adds the predicted stride to compute the final value. In addition to this, differential value history-based predictors maintain histories of strides instead of histories of values. The following are the advantages of strides over values:

- Some predictions that are considered as compulsory misses for values can be predicted using strides. For example, the value sequence 1, 3, 4, 6, 7, 9, 10, 12 can be represented using the strided sequence 2, 1, 2, 1, 2, 1, 2. Since the strided sequence is

more regular, it may possible for it to predict the next value in the sequence even though it has not seen it before.

- Some predictors apply learnings across contexts. For such predictors, strides can provide improved coverage due to positive aliasing. For example, PC x with value stream 1, 3, 5, 7, 9, 11 and PC y with value stream 101, 103, 105, 107, 109, 111 can both be represented using the strided stream 2, 2, 2, 2. It may be possible to predict values for PC y using the strided pattern learnt for PC x.
- Since multiple value streams can be represented by the same strided stream, it can cause storage efficiency thereby reducing predictor sizes. Hashing efficiency for strides may also be better than that for values since strides can usually be represented using fewer bits.

The following are some disadvantages of using strides over values.

- Strides can be less accurate than using values. For instance, the last two values of the value sequence 1, 3, 5, 7, 9, 10000, 150000 can be incorrectly predicted by a stride predictor that may learn a constant stride of 2 initially.
- For predictors that apply learnings across contexts, strided patterns can cause inaccuracy due to negative aliasing. For example, if PC x has a value pattern of 1, 2, 3, 4, 5, 6 and PC y has a value pattern of 100, 101, 102, 110, 135, a strided predictor may try to apply constant strides learnt from PC x on PC y. This will cause inaccuracy in predicting 110 and 135 for PC y.
- Stride based predictors require knowledge of the last observed value for a given context. This may not be available if the corresponding instruction has not completed execution. This can cause inaccuracy.

In summary, strides tend to provide more coverage than values. However, they may be more inaccurate. We wish to pick the best one for predictability purposes. We

evaluate them on the two value history-based predictors (PC Local Learning, PC Local Application and PC Local Learning, Global Application) and on the branch history-based predictor. The evaluation is done for different context lengths and using both oracle and practical divergence handling capability. The confidence threshold is fixed to 10.

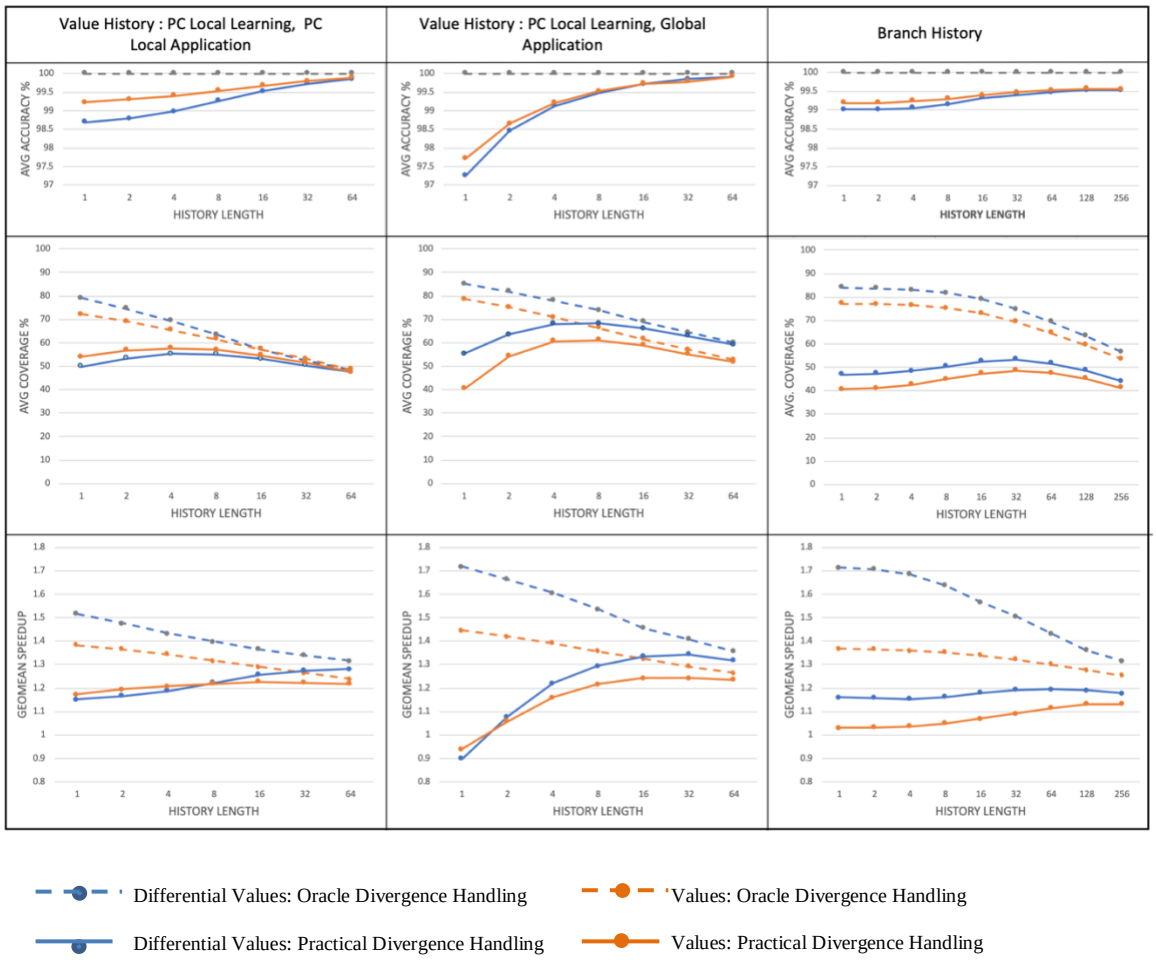

Figure 9: Evaluation of Differential Values vs Values

From Figure 9, we see that the accuracy for values is consistently better than that of differential values but only by a small margin. However, the overall coverage and speedup is mostly better for the latter than the former. Hence, we choose differential

values over values for our baseline. In the rest of this chapter, we mean differential values when we refer to values or value history.

3.2.2.8 Value History vs Branch History

In this section, we evaluate value history vs branch history. We first discuss the advantages of using value history over branch history:

- Value history-based prediction can capture any arbitrarily correlated value pattern for a given PC. Branch history-predictors can only capture constants or strides for a given PC and branch history.
- Intuitively, it seems more reasonable that previous values can be more direct indicators of the next value as opposed to branch history. In other words, predictability and coverage is expected to be better for value history.
- Value history-based predictors such as DFCM++ achieves better coverage than branch history-based predictors such as EVTAGE using much lower confidence counter widths.

The following are some disadvantages of using value history over branch history.

- Values are 64 bits wide and their histories have to be maintained on a per PC basis. Since PC local value histories are used to index into the prediction table, the number of entries in the predictor may also be large. However, there has been evidence on how only a small number of values are required to predict majority of the dynamic instructions [14]. Also, the DFCM++ predictor [8] has shown that is possible to compress 64 values into a single 64-bit value for predictor operation. On the other hand, storage complexity is somewhat simpler for branch history-based predictors as global branch history is used.

- Value histories can be incomplete if the corresponding producer instructions are still inflight. This can cause loss in accuracy and coverage. Value history-based predictors tend to maintain speculative histories to deal with this issue.

Value history-based predictors are further classified based on whether they apply PC localized learning globally (across PCs) or locally (within the same PC). We discuss tradeoffs between these two options.

- Often, PCs with similar value history tend to behave similarly. Global application helps capture this behavior. On the other hand, ability to learn patterns is affected when PCs with similar value history produce different values. PC local application works better under these circumstances.
- The predictor storage is reduced significantly if predictions are shared across PCs. This reduces the table size for a predictor with global application as opposed to one with PC local application.

Since branch history is global context information, we do not have a similar classification for it.

We evaluate the three predictors for different context lengths using both oracle and practical divergence handling. Figure 10, Figure 11 and Figure 12 depict the accuracy, coverage and speedup curves respectively.

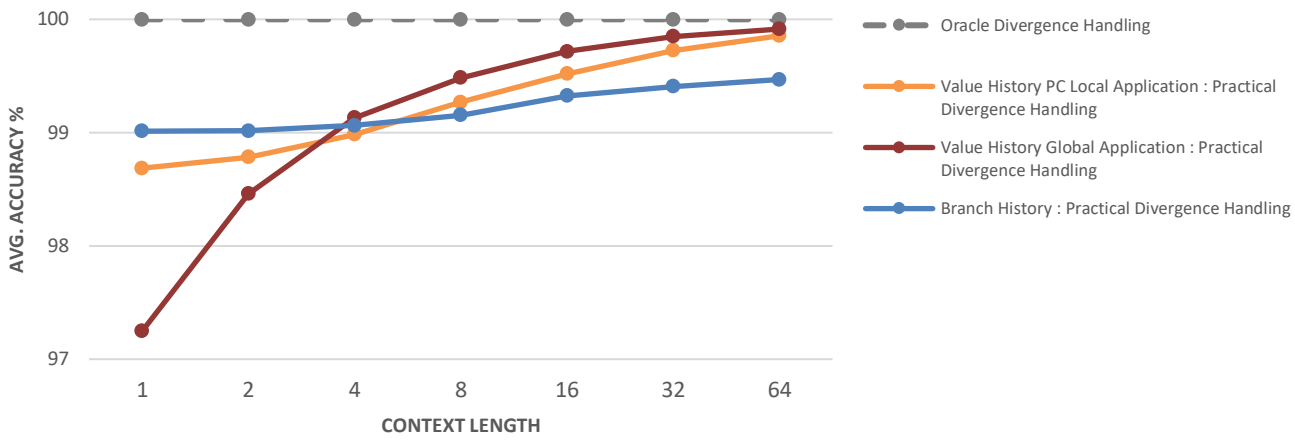

Figure 10: Accuracy Comparison for Predictors using different context information

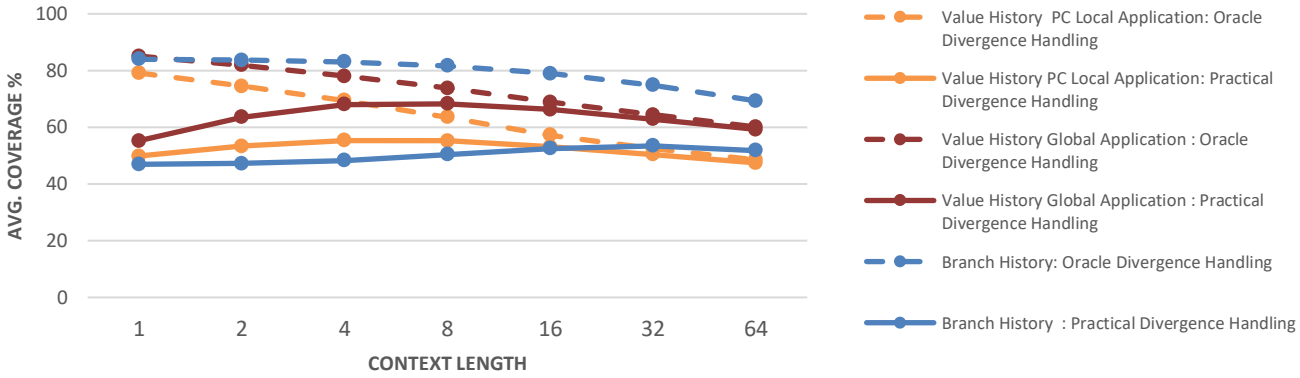

Figure 11: Coverage Comparison for Predictors using different context information

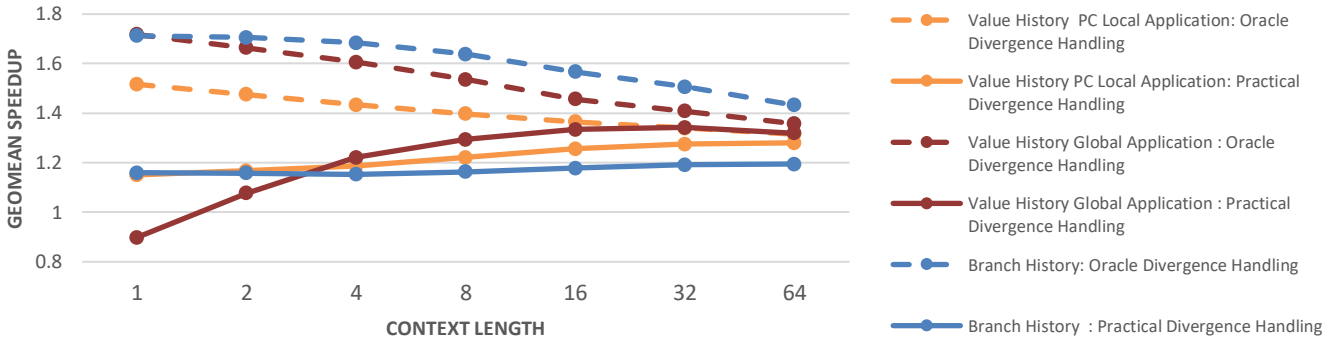

Figure 12: Speedup Comparison for Predictors using different context information

For value history-based predictors, we see that global application tends to perform better than PC local application for most of the context lengths. Hence, between the two, we pick the former.

The branch-history based predictor with oracle divergence handling capability has the best coverage and speedup indicating that it has potential for value prediction. However, we see that the branch history-based predictor with practical divergence handling capability performs the worst. In other words, branch history alone is not as powerful as value history is in detecting patterns as there is plenty of divergence that it can't handle. It may be possible to improve its accuracy and hence its speedup by using a higher confidence threshold than the one use for this experiment (i.e., 10). However, doing so will not equip it with additional divergence handling capability required to identify more patterns. It is only a defensive mechanism that avoids predicting for diverging values and results in lower coverage due to slower training.

For a predictor with practical divergence handling capability, value history with global application achieves the best accuracy, coverage and speedup for a context length of 4 or more. This is a strong indication that it provides better predictability than the others. Hence, we pick this as our baseline. Improving its performance without a drop in its coverage automatically means the resulting performance will be better than that of a branch history-based predictor (assuming oracle update).

3.2.3 Exploring Tradeoffs Across Value History Lengths

3.2.3.1 Accuracy vs Coverage

We delve deeper into the tradeoffs of using different context lengths in our baseline value history-based prediction method.

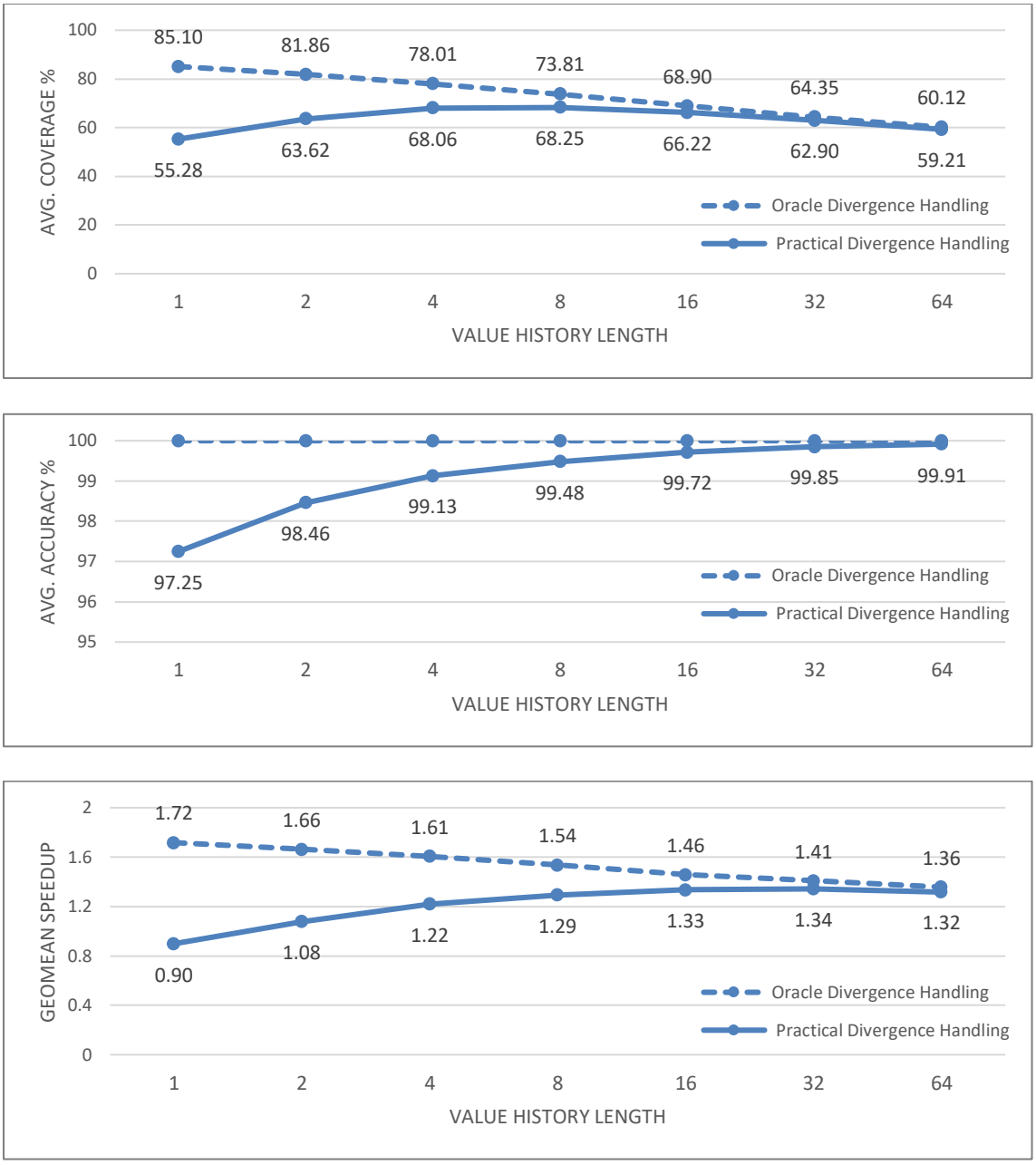

Figure 13: Accuracy, Coverage and Speedup for Different Value History Lengths

From Figure 13, we make the following important observations:

- The coverage curve for oracle divergence handling drops with increase in value history length. This is because it takes longer to train as context length increases. Assuming we can solve the divergence issue magically, this curve establishes the

upper limit for predictability using value history length alone. In other words, longer the value history, lesser is the scope for prediction.

- The coverage curve for practical divergence handling rises up to a length of 8 beyond which it falls off. As we increase value history length, we add more context information. This makes the predictor more intelligent allowing it to deal with diverging patterns. However, the training time increases with history length. As a net effect of the two, we see a rise followed by a drop.
- The accuracy curve for practical divergence handling rises with increase in value history length. This is partly because of the increased intelligence that longer value histories add, allowing the predictor to accurately identify diverging values that can be mispredicted otherwise. It is also because of the slower training that helps the predictor to remain on a defensive mode and not learn inaccuracies.
- The speedup curve for practical divergence handling reemphasizes the need for extremely high accuracy in value prediction. Although coverage drops beyond a length of 8, speedup peaks at a length of 32 where accuracy is high. This is despite the reduced prediction scope that exists at longer value history lengths. This also provides explanation for why a value history-based predictor like DFCM++ [8] operates at long history lengths of 32 and 64.
- Our key point is that there is plenty of opportunity at shorter value history lengths (<= 16) that is being unutilized. Firstly, the oracle divergence curve indicates that there is better prediction scope at shorter history lengths. Secondly, the coverage and accuracy gaps between the two curves indicate that there is significant divergence that is obstructing shorter value history lengths from achieving its full potential. Finally, while it may be more challenging to bridge the gap for very small lengths such as 1 or 2, value history lengths from 4 to 16 already have excellent coverage. Any means to

improve their accuracy without loss in coverage should result in better speedup than what value history length of 32 currently achieves.

Improving accuracy without hurting coverage cannot be done using defensive mechanisms such as increasing the confidence threshold. It requires more aggressive divergence handling capabilities. For instance, Figure 14 shows the effect of increasing the confidence threshold for a value history length of 16. While the accuracy improves, coverage drops resulting in lower speedup.

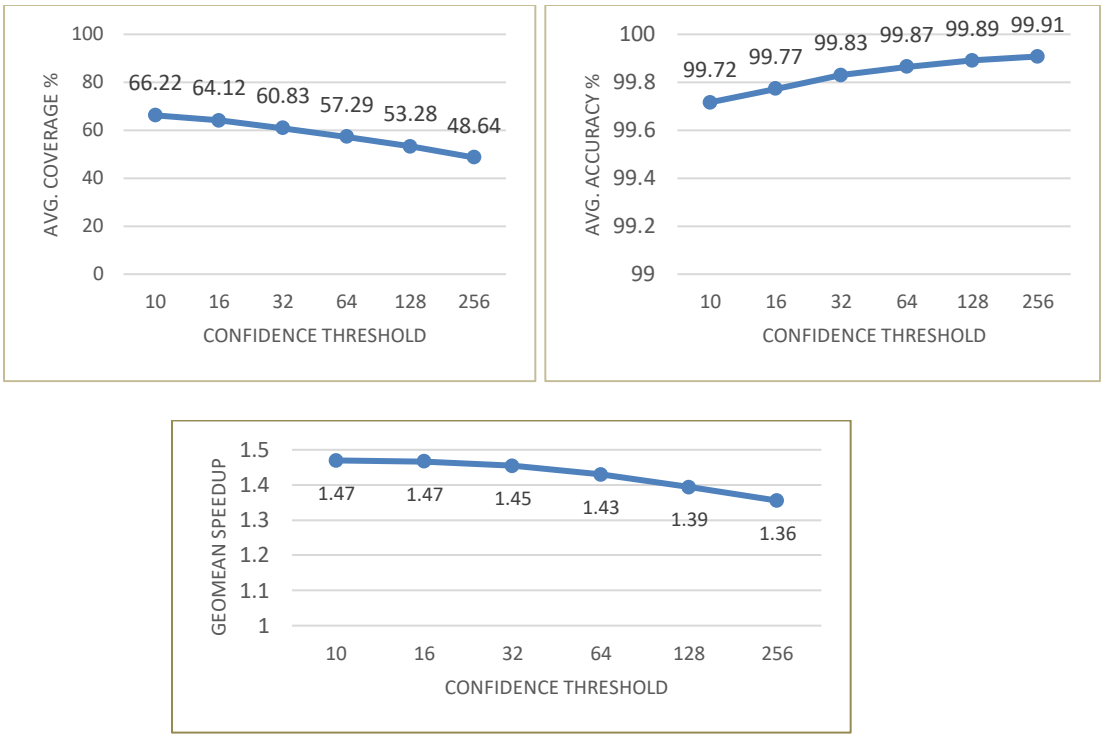

Figure 14: Effects of increasing confidence threshold for a value history length of 16

3.2.3.2 Divergence Handling Capability

In this section, we analyze how effective different value history lengths are in handling divergence. We need a worst-case baseline to establish the maximum possible

divergence that can occur. We then quantify how much of this divergence is handled by using different value history lengths.

It should be noted from Figure 13 that the coverage obtained for oracle divergence handling using a value history length of 1 establishes an upper bound to what can be predicted using any value history length. Predictions made using longer value history lengths are only a subset of what a value history length of 1 with oracle divergence handling can predict. To illustrate this, let us consider a value history length of 4 (with either practical or oracle divergence handling capability) that predicts the following correlated patterns correctly:

m a b c $\rightarrow$ x x is followed by a value stream consisting of m a b c

n a b c $\rightarrow$ y y is followed by a value stream consisting of n a b c

It will be possible to predict these patterns only if each of them have occurred enough number of times (>= threshold). This automatically means that the following have also occurred enough number of times (>= threshold).

c $\rightarrow$ x x is followed by a value stream consisting of c

c $\rightarrow$ y y is followed by a value stream consisting of c

Hence, with oracle divergence handling capability, the predictor with value history length = 1 will be able to predict both of these. In other words, everything that longer value history lengths predict is a subset of what value history length = 1 with oracle divergence handling capability can predict.

Note that a value history length of 2 with practical divergence handling may not be able to predict x and y correctly as the value stream is not long enough to identify the divergence. The context consisting of b c is followed by x sometimes and y sometimes.

b c $\rightarrow$ x x is followed by a value stream consisting of b c

b c $\rightarrow$ y y is followed by a value stream consisting of b c

In other words, longer value history lengths can provide additional context information that can help identify more divergences (at the cost of training time). In this respect, a history length of 1 with practical divergence handling identifies the least number of patterns and hence is worst performing.

With these points in mind, the coverage gap between the oracle and practical divergence handling curves with history length = 1 represents the maximum possible loss in coverage due to divergences. We consider this as the baseline and evaluate how much of this divergence is predicted using longer value history lengths.

Ability to handle divergences should equip the predictor with the following capabilities:

- Ability to increase coverage by predicting diverging values (handled divergence).
- Ability to reduce mispredictions that arise due to diverging value streams. This can be done either by predicting them correctly (corrected inaccuracy) or by not making any predictions for them (suppressed inaccuracy).

Terminology	**Outcome of Predictor being evaluated**	**Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1**	**Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1**
Handled Divergence	Predicted correctly	Predicted correctly	Mispredicted / not predicted
Coverage Loss	Not predicted / mispredicted	Predicted correctly	Predicted correctly

Table 1: Terminology describing coverage benefits and losses

Terminology	Outcome of Predictor being evaluated	Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1	Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1
Corrected Inaccuracy	Predicted correctly	Predicted correctly	Mispredicted
Suppressed Inaccuracy	Not predicted	Predicted correctly	Mispredicted
Inaccuracy	Mispredicted	Predicted Correctly	Predicted correctly / Not predicted / Mispredicted

Table 2: Terminology describing inaccuracies dealt / not dealt with

Table 1 describes the terminology used to quantify coverage benefits due to divergence handling and coverage losses due to increased training time. Table 2 describes the terminology used to quantify inaccuracies that are dealt with and inaccuracies that still remain.

Figure 15 shows the results of coverage evaluation. We see that the divergence handled increases initially with history length until it peaks for length = 8 beyond which it falls off. In other words, using very long histories don't bring the best divergence handling abilities as they are slower to train. Although history length = 32 had the best speedup due to better accuracy (Figure 13), history lengths between 4 to 16 handle more divergence. We are unable to utilize this in our predictors as the accuracy is not sufficient enough at these lengths. Also, as expected, we see that the coverage loss due to slower training increases with increase in value history lengths (Figure 15).

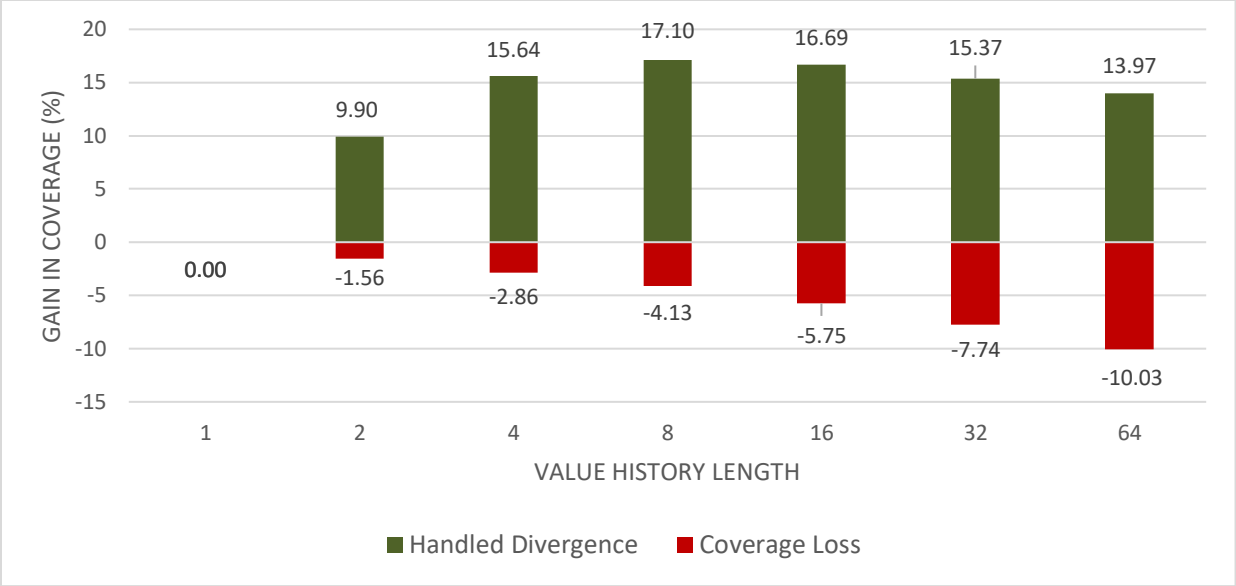

Figure 15: Value History: Coverage benefits and losses

Figure 16 shows the results of evaluation with respect to inaccuracies. We see that more inaccuracies are dealt with (i.e., corrected inaccuracy + suppressed inaccuracy) as value history length increases. The inaccuracy that remains also decreases with value history length. This explains why long lengths of 32 and 64 have the best overall accuracy. However, it should be noted that history lengths between 4 to 16 end up correcting more inaccuracies while longer lengths end up suppressing more of them. In other words, history lengths between 4 to 16 are more powerful in turning mispredictions into accurate predictions indicating better divergence handling capabilities. However, their overall accuracy is still relatively lower causing them to underperform.

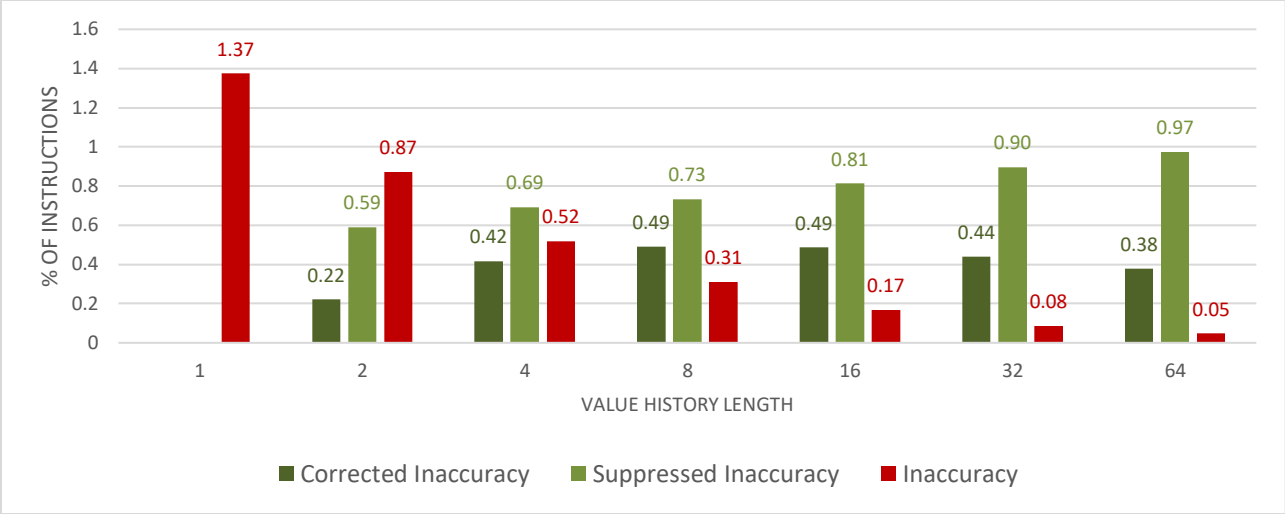

Figure 16: Value History: Inaccuracies dealt/not dealt with

In conclusion, value history-based prediction can be improved by one or more of the following methods:

- Equipping value history with better divergence handling capabilities so that they can perform closer to potential
- Improving accuracy, especially for history lengths = 8 to 16 without hurting their coverage to take advantage of divergence handling capabilities that they already have

3.3 Improving Predictability using Heterogeneous Context Information

3.3.1 Combining Branch History with Value History

We demonstrate that the combination of branch history with value history provides better predictability than using value history alone. It tackles some of the short comings described above. Particularly, we believe that the addition of branch history provides additional context information required for improved divergence handling.

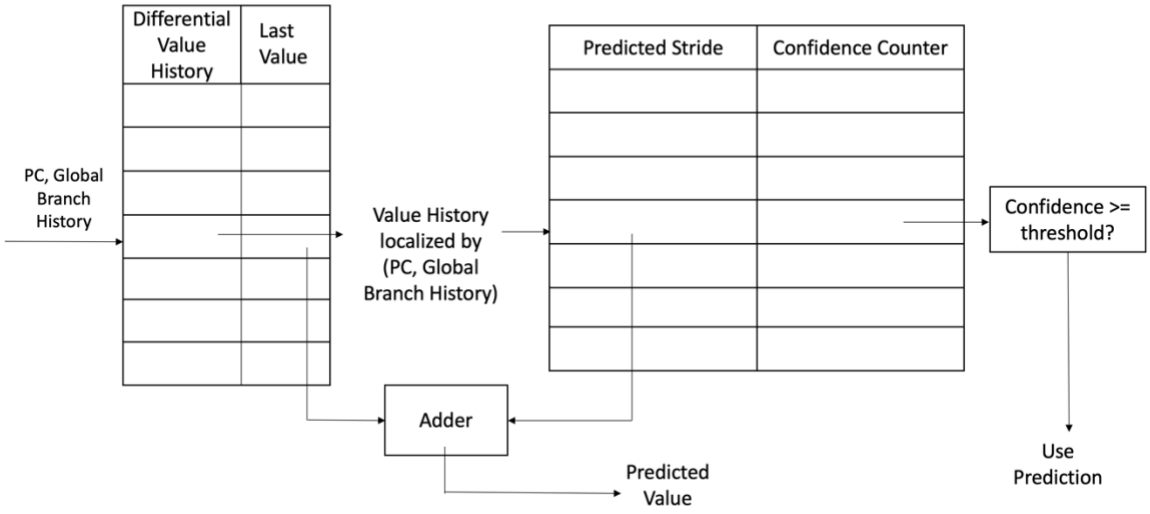

Figure 17: Predictor design that combines both Value History and Branch History

We design the predictor as shown in Figure 17. We index the first level table using a combination of PC and global branch history instead of using PC alone. Global branch history is a vector of 1s or 0s representing recent branch outcome history. All value histories are localized for a given (PC, global branch history). We use these histories to index into the second level prediction table.

An alternate design could have been to index the first level table using PC alone and to index the second level table using a combination of value history and global branch history. However, this method does not make much sense since the second level table is shared across PCs. We do not expect PCs to produce the same values as a result of similarity in branch history.

3.3.2 Evaluating Predictability

3.3.2.1 Accuracy vs Coverage

We discuss the impacts on accuracy (Figure 18), coverage (Figure 19) and speedup (Figure 20) when branch history is combined with value history. We empirically chose a branch history length of 128. A confidence threshold of 10 is used.

- From Figure 18, we see that the addition of branch history significantly improves accuracy across most value history lengths. This is an indication that branch history is able to prevent inaccuracies due to diverging value streams that are seen when value history alone is used.

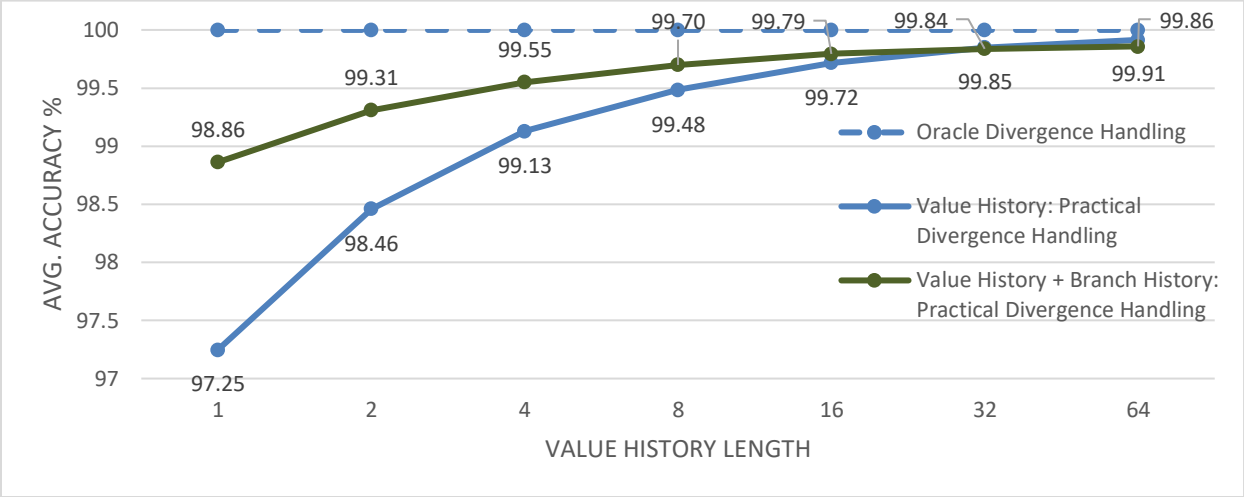

Figure 18: Accuracy comparison when branch history is combined with value history

- From Figure 19, we see that there is no significant drop in coverage despite adding more context information. Addition of branch history is expected to cause two issues.
    - Longer training times as a single PC localized value history stream is now split across multiple entries due to further localization by branch history
    - The splitting can also result in loss of predictability if a well correlated value pattern is now broken into multiple uncorrelated pieces

Yet, we see that the overall coverage does not drop significantly for any value history length. In fact, coverage improves significantly for shorter value history lengths of 1 and 2. The overall change in coverage varies from -1.42% to +8.40% across value history lengths. This is a strong indication that the addition of branch history is able to tackle additional divergence and/or make new predictions. This brings in additional coverage, thereby making up for any losses due to training overheads.

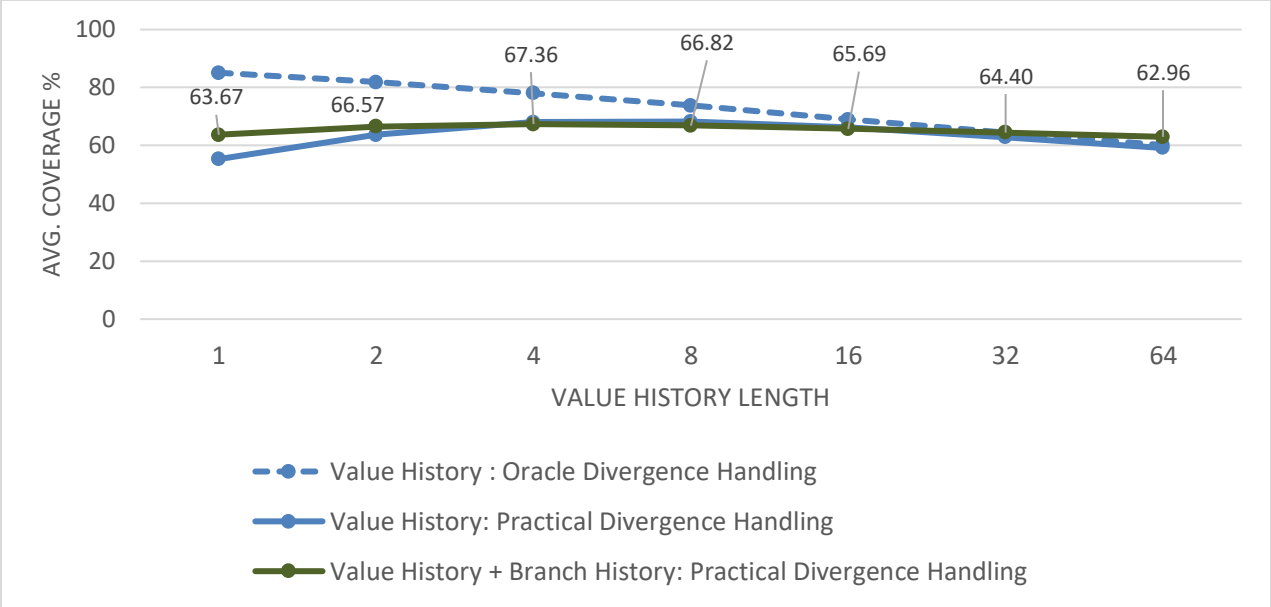

Figure 19: Coverage comparison when branch history is combined with value history

- From Figure 20, we see that the addition of branch history results in significant speedup over using value history alone. It peaks at 42% for a value history length of 16. Using value history alone achieves a peak speedup of only 34% at length = 32. Addition of branch history allows even a value history of length = 4 to exceed this performance by ~3%. Overall, better speedup can be attributed to better accuracy without compromising on coverage. Moreover, the increased accuracy ensures that

the coverage benefits at history lengths of 4 to 16 are utilized well resulting in better speedup at these lengths.

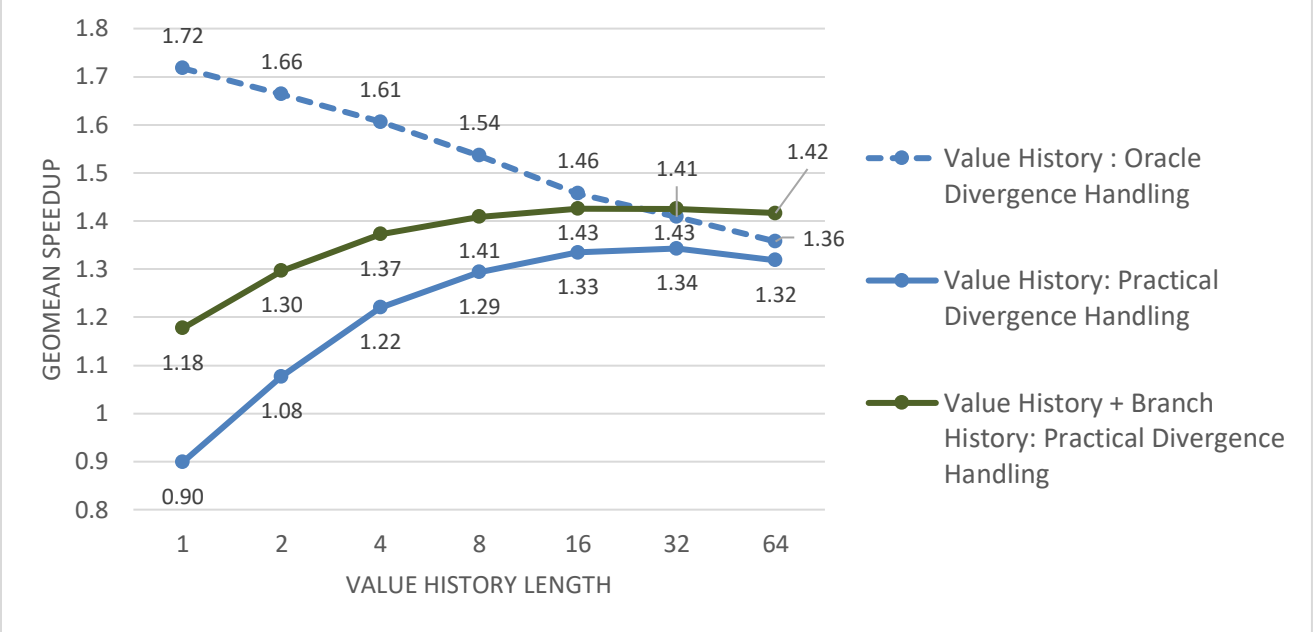

Figure 20: Speedup comparison when branch history is combined with value history

- From Figure 18, Figure 19 and Figure 20, we see that the addition of branch history lifts the curve for practical divergence handling closer to oracle divergence handling. This is a strong indication that divergence is being handled better. Another point to note is that the speedup slightly exceeds that of oracle divergence handling at very long value history lengths of 32 and 64. This is a possible indication that branch history is able to identify a few new correlated patterns that value history alone cannot identify. We quantify benefits due to these in the next section.

3.3.2.2 Divergence Handling Capability

We evaluate divergence handling capabilities for the combined use of branch history and value history as we did for value history alone in section 3.2.3.2.

For coverage and accuracy analysis, we use the same terminology as described earlier in Table 1 and Table 2. The addition of branch histories can discover new correlated patterns that even the oracle value history-based predictor of length 1 cannot. We introduce an additional term called "New Predictions" to quantify coverage gain due to such predictions (Table 3).

Terminology	**Outcome of Predictor being evaluated**	**Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1**	**Outcome of Predictor with Oracle Divergence handling capability and Value History Length = 1**
New Predictions	Predicted correctly	Not Predicted	Not Predicted / Predicted incorrectly

Table 3: Additional Terminology describing new coverage benefits

Figure 21 breaks down coverage differences into the following categories:

- Coverage gain due to handled divergence
- Coverage gain due to new predictions
- Coverage loss. This could be due to slower training. It could also be due to splitting of well correlated patterns into multiple uncorrelated pieces when branch history is added.

We compare for value history alone vs value history + branch history.

We see from Figure 21 that the addition of branch history improves divergence handling consistently. The additional coverage gains range from 1.1% to 8.7% across value history lengths. It also provides a small coverage gain of ~1.4% from new predictions. On the downside, there is increased coverage loss due the overheads of adding more context information. The coverage loss becomes worse by about 0.5% to

8.8% across history lengths when compared to using value history alone. In summary, coverage losses are unavoidable due to the overheads of increased context information. However, the addition of branch history is quite powerful in handling divergences and identifying patterns that the gains make up for the loss. As a result, the overall coverage does not drop significantly for any value history length (Figure 19).

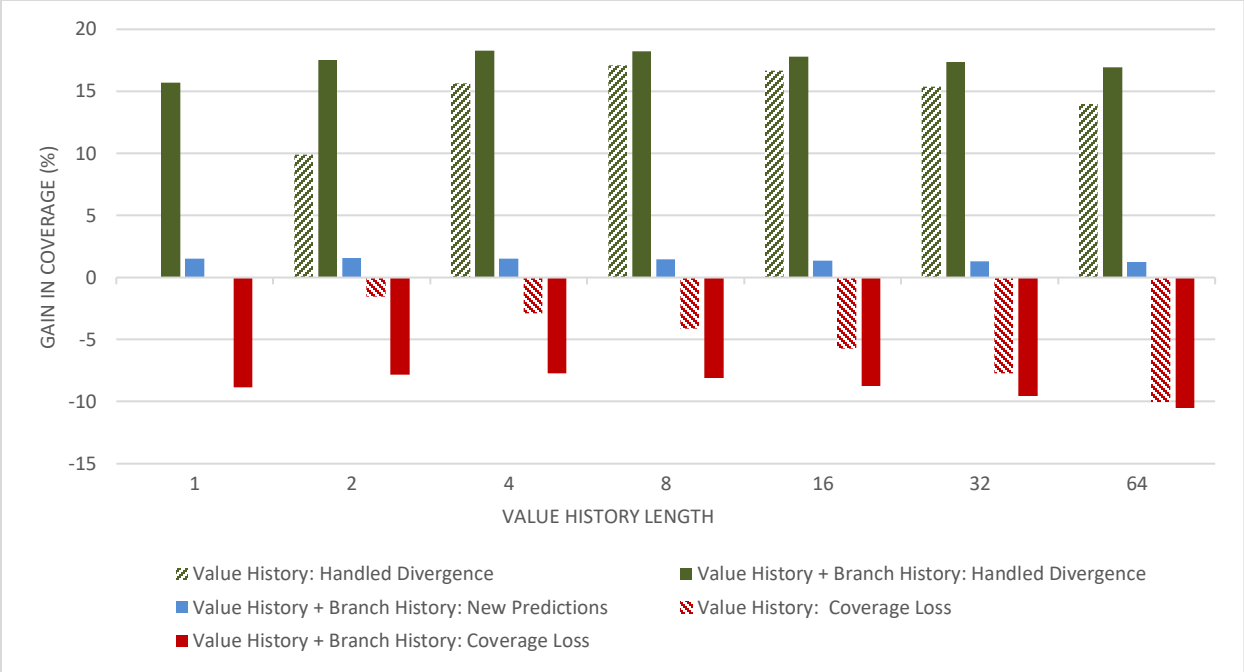

Figure 21: Comparison of coverage benefits and losses

Figure 22 analyzes accuracy. If we sum up corrected inaccuracy with suppressed inaccuracy, we see that the addition of branch history is able to handle more inaccuracies as opposed to using value history alone. It handles about 0.02% to 1.22% more inaccuracy than the latter. We also see that branch history + value history corrects majority of the inaccuracies while value history alone suppresses majority of them. This is again indicative of the more powerful divergence handling capability

that addition of branch history brings. Besides, we see that the inaccuracy that continues to remain is lower by about 0.0% to 0.8% when branch history is added as opposed to using value history alone.

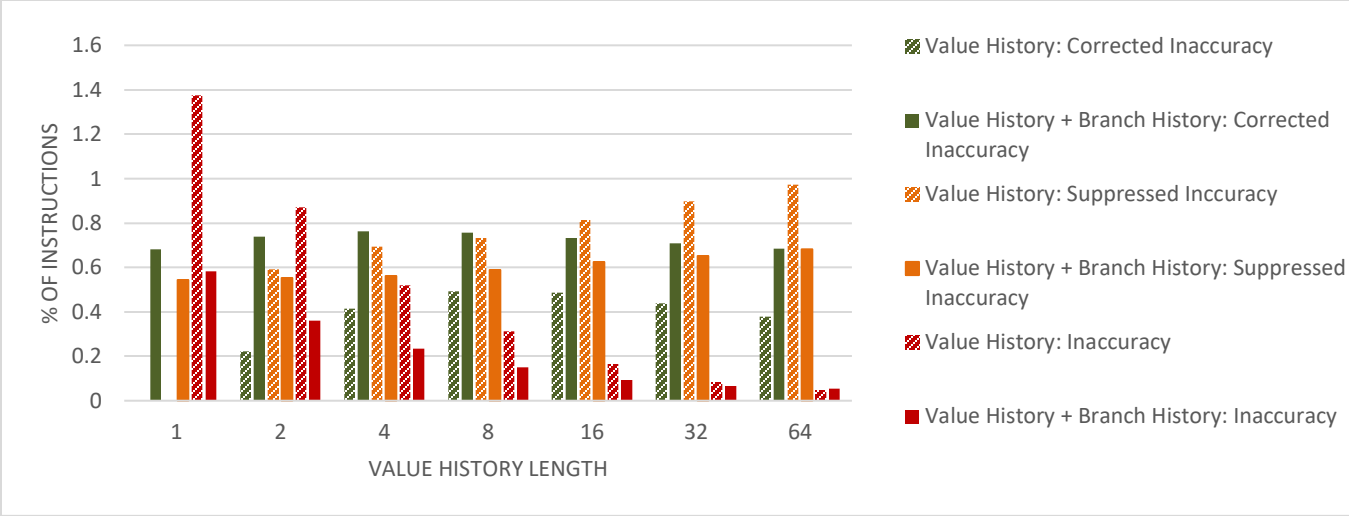

Figure 22: Comparison of inaccuracies dealt/not dealt with

In summary, predictability and accuracy improves significantly by combining the use of value history with branch history. This translates into better performance overall.

3.4 THE UPDATE PROBLEM

So far, we assumed an oracle update mechanism where the result of a prediction is known immediately after making the prediction. We now move to a more realistic update scheme where there will be a delay of many cycles between prediction and the corresponding update. Also, from this point, we only talk about practical divergence handling. There are no more oracle mechanisms in the predictor making it fully realistic.

Any value prediction mechanism that uses value history suffers from the update problem. Value histories that are stored in the table can become stale if some of the

instructions for the corresponding entry are in flight. This makes the result of prediction dependent on the outcomes of the instructions in flight. This issue is encountered when the same static instruction is scheduled back to back because of tight loops.

We study the impact of the update problem on the performance of both types of predictors – named value history based and value history + branch history based. For value history alone, we use a history length of 32. When combined with branch history, we use a value history length of 16.

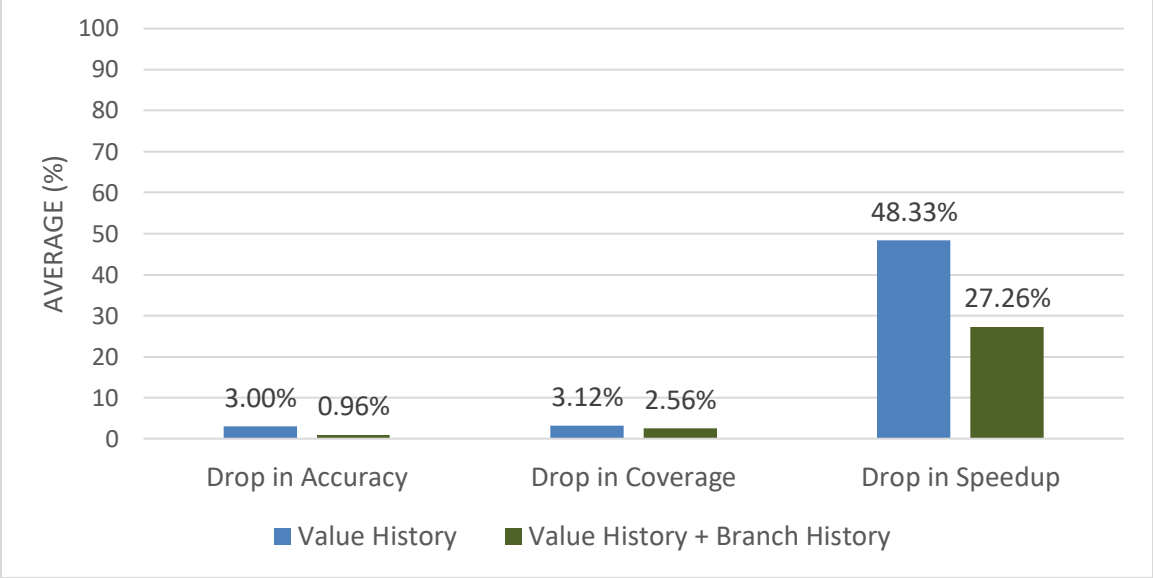

Figure 23: Drop in performance when the oracle update mechanism is replaced with an actual update

Figure 23 depicts the drop in accuracy, coverage and speedup when the oracle update scheme is replaced with an actual update scheme. Clearly, a value history-based predictor is severely affected by the update problem. Its accuracy drops by an unacceptable 3%. Its speedup takes a beating of 48% resulting in an overall speedup that is negative (-15%). On the other hand, combining value history with branch history makes it relatively more tolerant to the update problem. Its accuracy drop is relatively

smaller (~0.96%). A speedup drop of ~27.3% still ensures overall positive speedup (~15%).

When using value history alone, predictions to the same PC close to one another is a concern. When using value history combined with branch history, predictions to the same PC + global branch history close to one another is a concern. We evaluated our traces to find that about ~37% of the times, at least one previous instance of the same PC as the one currently being predicted is in inflight. However, only about ~20% of the times, an instruction with the same PC + global branch history is in inflight. This explains why a combination of value history with branch history is more tolerant to the update problem than using value history alone.

To tackle the update problem, predictors that use value history alone need to maintain speculative histories in addition to actual histories. These speculative histories are constructed at least partially using predicted values instead of actual values. While it is a common practice in branch prediction to maintain speculative branch histories, the same does not work equally well with respect to value history. For both cases, when predictions are correct, speculative histories work fine. When predictions are wrong, any instruction that is fetched after the mispredicted instruction is flushed. As a result, it does not matter if a wrong speculative history was used to predict for it. However, in value prediction, there is an additional complexity when a prediction is not made due to lack of confidence. As a branch prediction is always made, this issue does not exist for branch prediction or for value prediction using branch histories. When value prediction confidence is low, it may not be a good idea to use the value to construct a speculative value history as doing so can cause mispredictions for instructions that use the history. Hence, the idea of using speculative value history is not perfect and it cannot match the performance of an oracle update mechanism.

Also, there is significant hardware storage required to maintain speculative histories. For example, the DFCM++ predictor uses two value histories – of lengths 32 and 64 respectively. For each entry in the table, 4 histories are maintained including 2 that are speculative. Updating speculative histories also increases the complexity of the core. It is possible that committed instructions update the history at the same time that fetched instructions use it for prediction. In that case, either forwarding needs to be enabled or one of two have to be deprioritized.

A simpler alternative to speculative histories is to simply check if any of the instructions in flight have the same signature (PC or PC + global branch history) as that of the instruction being predicted. If there is, we skip predicting for the instruction. Although this mechanism will result is some coverage drop when compared to maintaining speculative histories, it will result in better accuracy. Figure 24 shows us the results for evaluation of this mechanism. We see that that combined use of value history with branch history gives significantly more speedup than using value history alone when this mechanism is used to deal with the update problem.

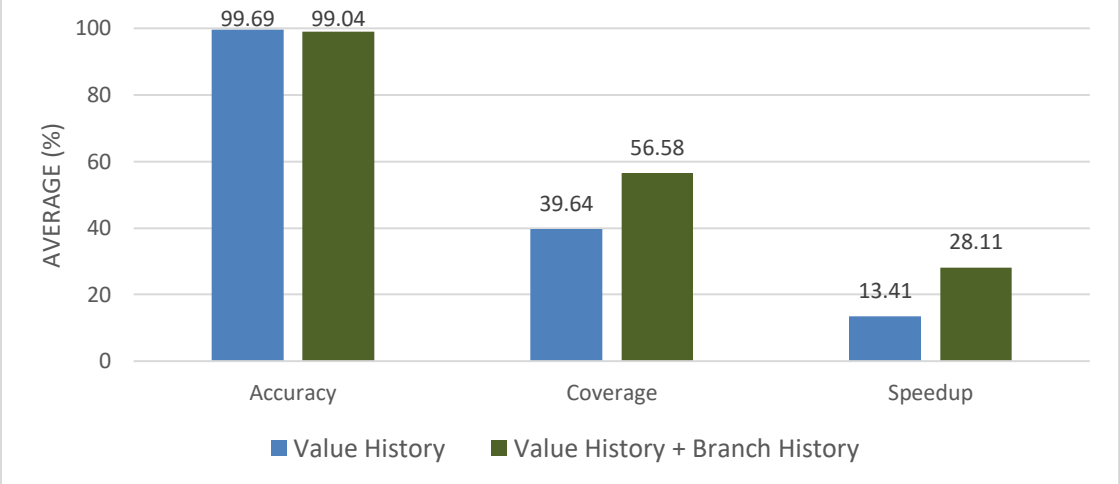

Figure 24: Overall performance when predictions are not made for instructions that share the same signature with one more inflight instructions

Thus, we conclude that the combination of branch history with value history can achieve better performance than using value history alone. It adds better predictability and is more tolerant to the update problem. We call our predictor (Figure 17) as the Heterogeneous Context-based Value Predictor (HCVP). We use an update policy that skips predicting for instructions that share signatures with one or more instructions in flight.

3.5 EVALUATION

3.5.1 Methodology

We evaluate HCVP using the simulation infrastructure released by the First Championship Value Prediction (CVP-1). It models a 16 wide out-of-order processor with a large instruction window (256 entry ROB), perfect branch prediction and unlimited number of functional units. It uses a large value misprediction penalty with a complete pipeline flush at commit time.

For evaluation, we use the 135 public traces from Qualcomm Datacenter Technologies released along with the CVP-1 infrastructure. It includes compute and memory intensive workloads, as well as a large number of server class traces [7].

3.5.2 Comparison with Other Predictors

We evaluate HCVP against the winning entries from the unlimited category of CVP-1, namely Enhanced VTAGE Enhanced Stride (EVES) and DFCM++. Both these predictors are hybrid predictors that combine context-based components with computational predictors. While EVES combines a branch-history based predictor with a stride predictor, DFCM++ combines a value history-based predictor with a value

estimator. We first evaluate HCVP against the individual components of these predictors.

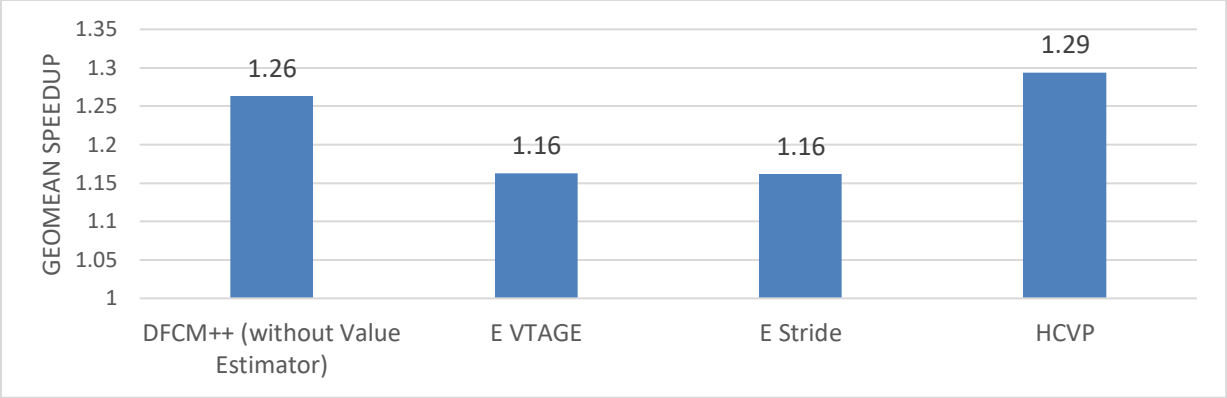

Figure 25: Speedup comparison across standalone predictors

Figure 25 shows the speedup comparison. HCVP outperforms the other predictors by achieving a geomean speedup for 1.29. Closest is DFCM++ with a geomean speedup of 1.26. Note that HCVP is able to outperform both a value history-based predictor (DFCM++) and a branch history-based predictor (E VTAGE). This is despite the fact that both these predictors use many features that HCVP is being evaluated without. These features can be applied to HCVP to improve its performance further. Some of them are listed below:

- While DFCM++ and E VTAGE use multiple context lengths and choose from them dynamically, HCVP uses a single context length.
- DFCM++ maintains speculative histories to deal with the update problem whereas HCVP does not.
- E VTAGE tunes its confidence threshold dynamically based on expected performance gain per prediction. HCVP uses a fixed confidence threshold.
- DFCM++ uses a PC Blacklister to reduce inaccuracies while HCVP does not.

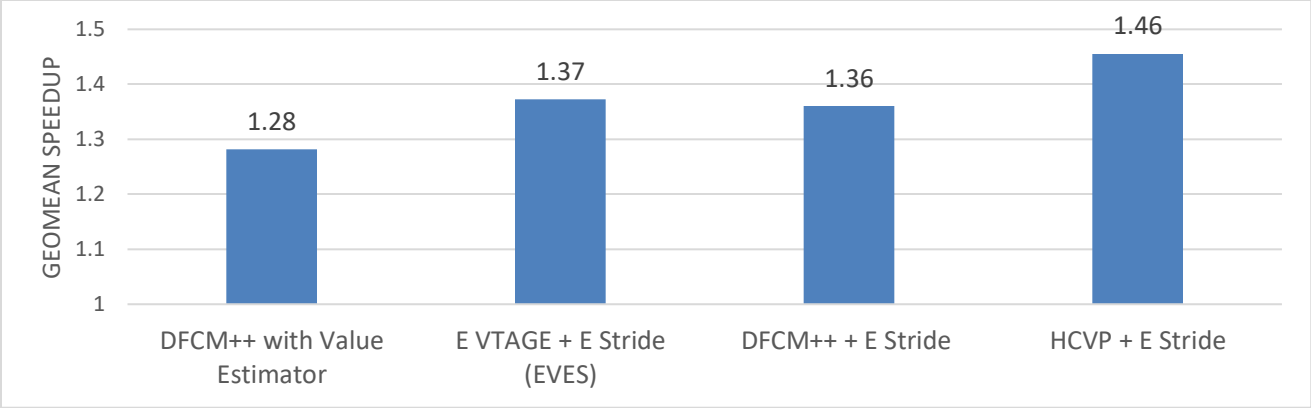

Figure 26: Speedup comparison across hybrid predictors

Figure 26 shows the speedup comparison across hybrid predictors. HCVP is combined with the Enhanced Stride predictor. The Enhanced Stride predictor is a small structure of less than 1KB that can provide a significant 16% speedup by itself. In the combined predictor, we prefer to use the predictions of HCVP over E Stride when both of them make confident predictions. We see that HCVP + E Stride outperforms EVES (the winner of CVP-1) by ~9%.

3.5.3 Sensitivity to Value History Length

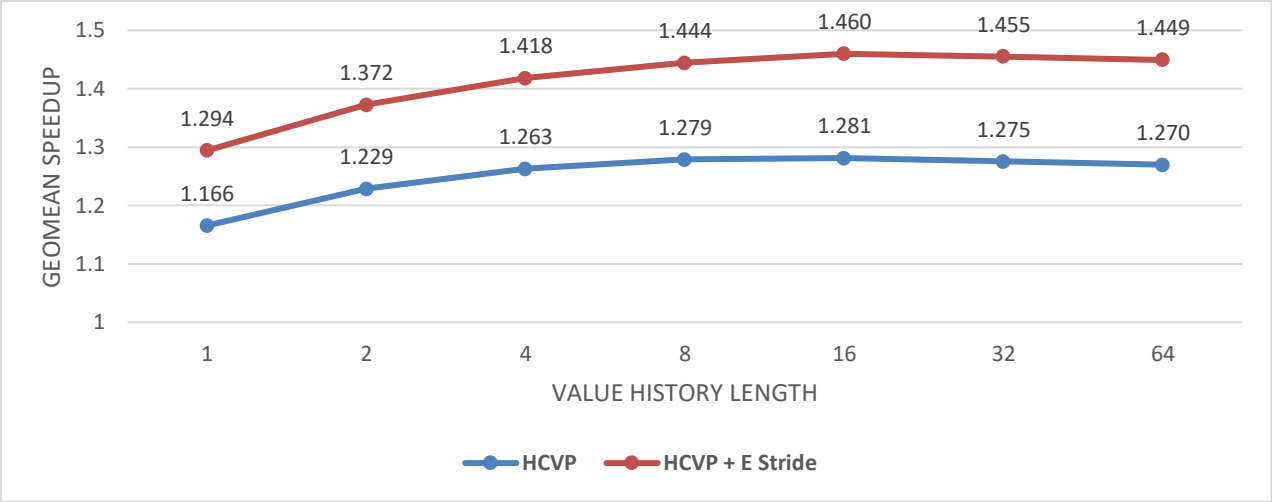

Figure 27: HCVP Speedup comparison for varying value history lengths

We evaluate the sensitivity of HCVP to value history length by fixing the branch history length to 128 and confidence threshold to 10. From Figure 27, we see that HCVP achieves a peak speedup of 28% at a value history length of 16. However, performance decreases by less than 2% even if we reduce history length to 4. We also show the performance variation for the hybrid predictor consisting of HCVP and E Stride. We see that a value history length as small as 2 is sufficient to match the performance of EVES (37%).

3.5.4 Sensitivity to Branch History Length

We evaluate the sensitivity of HCVP to branch history length by fixing the value history length to 16 and confidence threshold to 10. From Figure 28, we see that peak speedup is achieved at a branch history length of 128. Performance drops steadily with shorter lengths. This indicates that long branch histories are required to capture program context well.

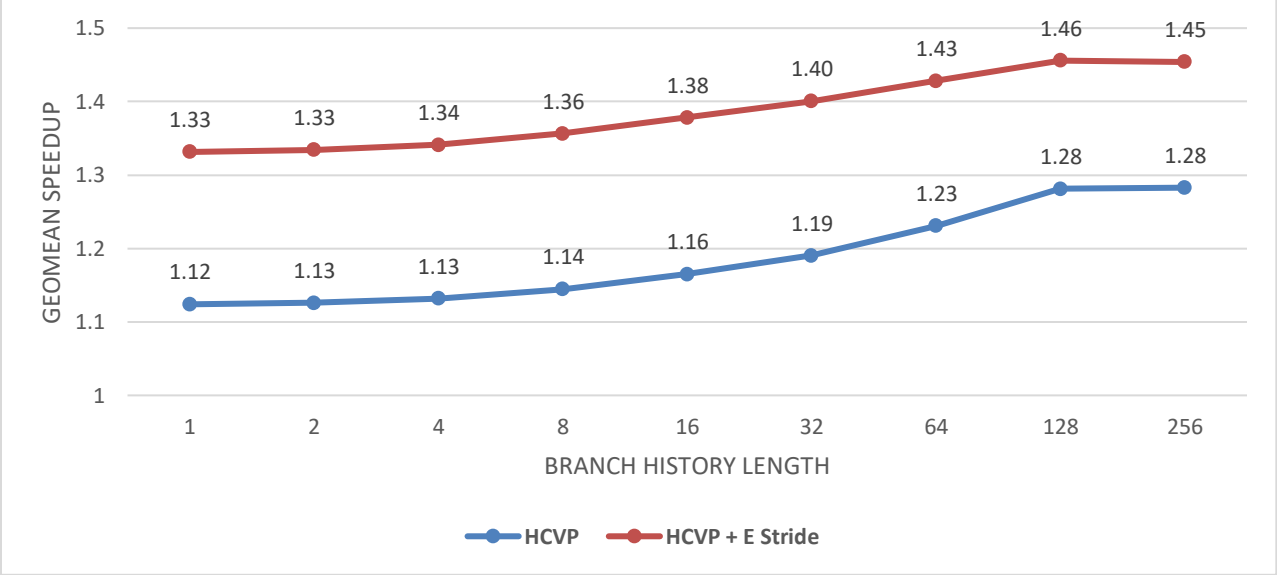

Figure 28: HCVP Speedup comparison for varying branch history lengths

3.5.4 Sensitivity to Confidence Threshold

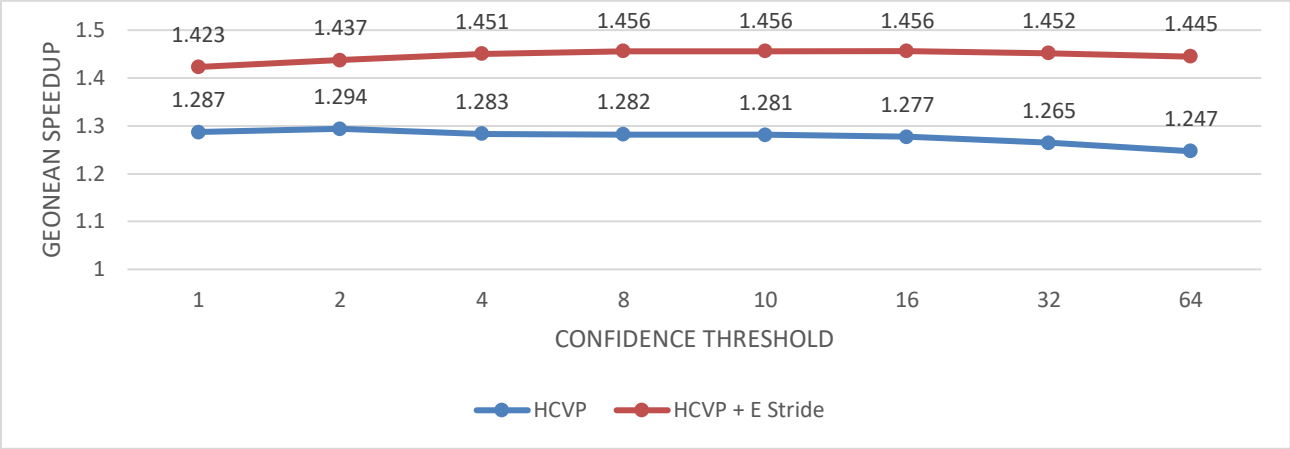

Figure 29: HCVP Speedup comparison for varying confidence thresholds

We evaluate the sensitivity of HCVP to confidence threshold by fixing the value history length to 16 and branch history length to 128. From Figure 29, we see that our predictor is not very sensitive to confidence thresholds. While HCVP achieves its peak speedup at a confidence threshold as low as 2, the hybrid that combines HCVP with E Stride achieves its peak speedup at 8. This is a significant departure from previously established ideas about value prediction requiring high confidence thresholds of 64-256 [5] [3]. Further, from Figure 30, we see that there is minimal drop in coverage when the threshold is increased. This provides a strong indication that contexts are followed by the same values consistently. Otherwise, the repeated re-training overheads should have caused a steeper drop in coverage with increasing thresholds. In summary, the context representation used is extremely powerful at identifying correlated patterns to an extent that confidence mechanisms are not that crucial anymore.

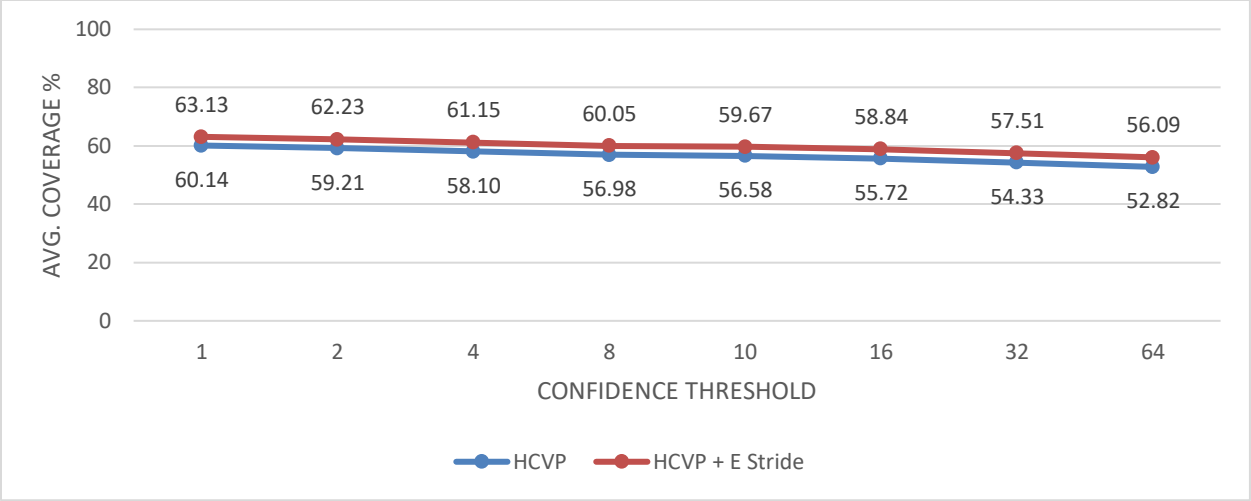

Figure 30: HCVP Coverage comparison for varying confidence thresholds

3.5.6 Hardware Complexity

Although our evaluation assumes that unlimited storage is available, we have a few points on why the HCVP predictor is inherently simpler than the other predictors.

- HCVP achieves the best performance even though it uses a fixed context length. It requires only two tables – a first level table to store histories and a second level table to store predictions. In contrast, the E VTAGE predictor uses multiple tables for multiple context lengths. DFCM++ uses two history lengths and hence has two second level tables. Further, it maintains preference counters for every entry in its first level table to choose between the two second level tables.
- HCVP outperforms DFCM++ without maintaining speculative histories. This reduces both storage and predictor operation complexity.
- HCVP performs well with small value histories in the 2 to 16 range as opposed to DFCM++ that requires 32 to 64 values per history. Smaller value histories can be compressed into fewer bits and/or can achieve better hashing efficiency.
- HCVP requires small confidence counters as the thresholds used are extremely small (2 to 8).

On the downside, since HCVP uses a combination of PC and value history, it is possible that the number of entries required in the first level table increases when compared to that of DFCM++.

3.6 CONCLUSIONS

In summary, we have demonstrated that a combination of value history with branch history is a better predictor of values than either of them used individually. Our evaluation shows that this combination provides additional divergence handling capabilities and accuracy when compared to using value history alone and that it allows for the use of mid-length value histories that provide better coverage than very long ones. Additionally, we have deemonstrated how this combination is relatively more tolerant to the update problem and hence achieves good performance even without maintaining speculative histories. The HCVP predictor designed based on these principles outperforms the winners of the First Championship Value Prediction.

3.7 FUTURE WORK

As part of future work, predictor table sizes need to be limited to demonstrate that that HCVP is practically viable. Further, HCVP is a naïve implementation of the idea that a combination of value history with branch history can work better than either of them used individually. However, there is plenty of opportunity to improve its performance by applying generic ideas that other state-of-the-art predictors employ. This includes the use of variable context lengths, dynamic adjustment of confidence thresholds and PC blacklisting to name a few. Further, The E VTAGE predictor uses information related to both branch targets and branch outcomes to construct its branch history. However, in our

current implementation, HCVP uses only branch outcomes. There is potential room for improvement if branch target information is also taken into account.

While the use of heterogenous context information has been demonstrated to be powerful for value prediction, it can be explored for other hardware prediction problems such as address prediction or prefetching as well.

Chapter 4: Maximizing Value Prediction Gains

4.1 INTRODUCTION

Fundamentally, value prediction is a technique that improves instruction level parallelism. It breaks true data dependencies by allowing early execution of dependent instructions that use the predicted value as a source value. In the absence of a value predictor, dependent instructions stall until their source values are ready. This can affect performance in multiple ways. Stalling of instructions can create critical paths, preventing forward progress in program execution. It can cause underutilization of hardware execution resources during the stall and an excess contention for the resources once the source values are ready. It can also prevent additional instructions from flowing through the pipeline if the fetch unit stalls as a result of the buffers filling up. Due to the large number of factors involved, the number of cycles saved due to a single correct prediction is highly variable and difficult to quantify. For example, performing value prediction on a low latency instruction may bring no benefit at all if the core has enough work to do to hide the latency. On the other hand, performing value prediction on a high latency instruction with a long dependent chain of instructions may save significant number of cycles.

Incorrect predictions involve performance penalties. These penalties are undesirable for the following reasons:

- If we consider the baseline to be a system that does not have a value predictor, a wrong prediction can result in performance that is worse than the baseline performance. This is a key difference between value prediction and branch

prediction. Predicting a branch is necessary to determine the instruction address for the next fetch and to ensure continued instruction flow through the pipeline. However, a value prediction may not necessarily be required to keep the pipeline occupied as there may be other instructions that are ready to be fetched and/or executed.

- The performance penalties associated with value mispredictions can be significant. Based on the recovery mechanism used, a single misprediction can cost anywhere between 5 to 50 cycles [1].

Keeping these in mind, the obvious goal for a value predictor is to be able to extract high performance gains from correct predictions so that it outweighs the severe performance degradation that can occur due to mispredictions. Speedup can be maximized by achieving one or more of the following:

1. Maximizing performance benefit per correct prediction
2. Maximizing the number of correct predictions
3. Minimizing the number of mispredictions
4. Minimizing performance penalty per mis predicted instruction

This section focuses on maximizing the performance benefit per correct prediction. We categorize instructions by various properties in an attempt to identify one or more classes of instructions whose values are highly beneficial to predict.

To illustrate that the average benefit per correct prediction varies largely based on the instructions being predicted, we perform perfect value prediction on 10% of the prediction-eligible instructions in a sample trace. Perfect value prediction is an oracle mechanism that assumes that the correct value is known during prediction time. In other words, it operates at 100% accuracy. We repeat the experiment multiple times by varying the 10% of the instructions that are chosen. From Figure 31, we see that there are large

performance variations across the runs, indicating that it is more beneficial to predict some values over the others. This brings us to the concept of "criticality". Some instructions are considered "critical" i.e., they lie in the critical path of execution. Any mechanism used to improve ILP provides maximum gains when they are applied to these critical instructions. For example, the concept of prioritizing critical instructions over others have been explored for cache management and prefetching [2]. In the context of value prediction, it may be highly beneficial to predict values produced by instructions that are on the critical path while it may be of no use to predict ones that aren't.

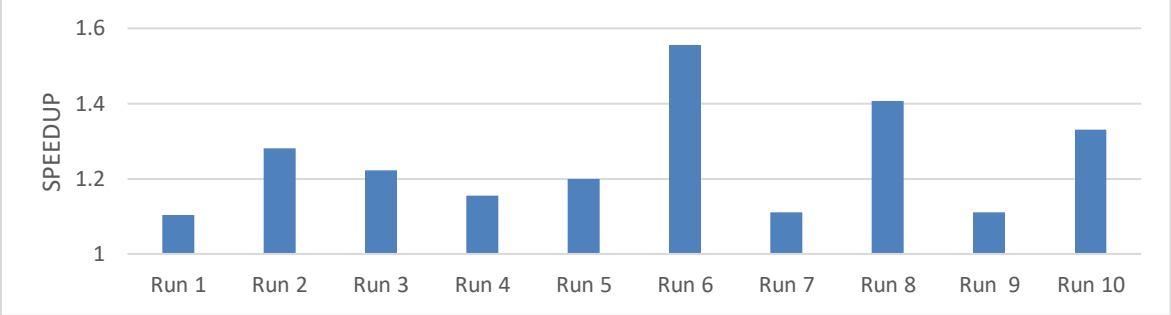

Figure 31: Variation in speedup by applying perfect value prediction on 10% of the instructions chosen randomly

Identification of critical instructions has often been considered as a difficult problem. It involves analyzing the execution at runtime to identify critical paths. Optimizing for performance on the critical path can cause another path to become the new critical path. Runtime identification and analysis of these paths can involve overheads in performance, power and cost.

A simpler alternative can be to identify (statically) classes of instructions that are likely to be on the critical path and are hence likely to provide significant benefit per correct prediction. If we are able to identify one or more such classes, it can influence the

way value predictors are designed. Particularly, we believe that it can have the following influences:

1. To illustrate the severe imbalance between the average benefit per correct prediction and the average loss in performance per misprediction, we perform two experiments on a sample trace by choosing 10% of its instructions randomly. In the first experiment, we perform perfect value prediction on the chosen instructions. In the next experiment, we mispredict for all of them. We observe that the speedup due to correct predictions is only 6.5% while the loss in performance due to incorrect ones is -45.3%. To deal with this imbalance, value predictors have focused on achieving very high accuracy at the cost of coverage. Often, they aim for an accuracy of 99% or above. We think the accuracy requirements can be relaxed further if predictors focused only on instructions that provide high benefits for correct predictions. This will allow them to adopt a less conservative approach that aims for higher coverage (within the class of instructions being targeted) at relatively lower accuracy.
2. Alternately, predictors can be designed to target all prediction eligible-instructions, but can apply different strategies based on the class of instructions being targeted. They can apply less accurate strategies for instructions that are more likely to be critical and vice versa. For example, the Enhanced VTAGE predictor varies the threshold for prediction confidence based on how much benefit predicting a particular instruction's value may bring [3].
3. Targeting specific classes of instructions can open up the possibility of designing prediction mechanisms that work specifically for the class in consideration. For example, there has been work on predicting addresses instead of values for

predictors that target load instructions alone. The value can be looked up in the cache using the predicted address [4].

4. Choosing a misprediction recovery mechanism often involves making a tradeoff between hardware complexity and performance. For example, pipeline squashing is a mechanism where all instructions younger than the mispredicted instruction are re-fetched and re-executed. While this mechanism is simple to implement, it has high latency since instructions that are independent of the mispredicted instruction are also re-fetched and re-executed. Selective replay is an alternate mechanism where only the dependent chain of instructions are re-executed. It is complex to implement as it requires runtime identification of the dependent chain. But it may have lower latency owing to the fact that independent instructions are not re-executed. Value prediction that targets only critical instructions can have increased tolerance to longer misprediction penalties. This can allow one to choose a mechanism like pipeline squashing that is simpler to implement in hardware although it has higher misprediction penalties associated with it.
5. The predictor state that needs to be maintained can be reduced if only a small fraction of instructions are to be tracked. In the best case, we hope to find classes of instructions that are only a small fraction of the total number of prediction-eligible instructions but provide high speedup.

4.2 METHODOLOGY

We categorize instructions by various parameters and study the speedup obtained by performing perfect value prediction on them. To evaluate the different classes of instructions, we use three metrics:

1. Speedup = IPC when all instructions of the given class are predicted / IPC when no value prediction is applied

   Avg. Speedup is the geomean speedup across all benchmarks. This metric quantifies the speedup that can be obtained if perfect value prediction is applied on all instructions that belong to the class in consideration.

2. Coverage = Number of Instructions that belong to the Class / Total Number of Value Prediction Eligible Instructions

   This metric quantifies the maximum coverage that can be obtained if we predicted all instructions that belonged to the class correctly. Avg. coverage is the arithmetic mean of coverage across all the benchmarks. High coverage means more predictor state to maintain.

3. Class Criticality = Speedup Percentage / Coverage Percentage

   Some classes of instructions may have high avg. speedup as a result of having high coverage, but the benefit per correct prediction may still be low. For a fair evaluation of the relative merit in predicting instructions of different classes, we need a better approximation for the benefit per correct prediction. We define class criticality as the average of the ratio between speedup % and coverage %. Avg. class criticality is computed as the arithmetic mean of class criticality across benchmarks

We run benchmarks twice to perform the experiments. In the first run, we collect data on various instructions. This data includes execution result, execution latency, instruction type (opcode) and instruction fanout to name a few. In the second run, we selectively target specific instruction classes using the data collected previously. We evaluate the benefits of value prediction using the three metrics specified above.

We use the simulation infrastructure released by the First Championship Value Prediction (CVP-1). It models a 16 wide out-of-order processor with a large instruction window (256 entry ROB), perfect branch prediction and unlimited number of functional units. It uses a large value misprediction penalty with a complete pipeline flush at commit time.

For evaluation, we use the 135 public traces from Qualcomm Datacenter Technologies released along with the CVP-1 infrastructure. It includes compute and memory intensive workloads, as well as a large number of server class traces [7].

4.2 RESULTS

4.2.1 Classification by Latency of Instruction Execution

In this section, we classify instructions by their execution latency and quantify the benefits of perfect value prediction on each one of the categories. We believe this is worth pursuing as long latency instructions are likely to be more beneficial for value prediction. The execution latency of the producer instruction will have a direct effect on the number of cycles dependent instructions stall for. A long latency instruction can cause dependent instructions to stall for many cycles. It can also cause a long dependent chain of instruction waiting to be executed.

For this experiment, Table 6 specifies how instructions have been classified based on their execution latencies. Table 4 contains the minimum and maximum latencies for instructions with different opcodes. Table 5 contains latencies for different memory accesses as specified for the CVP Simulator. The classification in Table 6 is based on these two.

Instruction Type	Min. Latency (cycles)	Max Latency (cycles)

ALU	1	1
Loads	2	150+
Stores	1	1
Unconditional Direct Branches	1	1
Unconditional Indirect Branches	1	1
FP	3	3
Slow ALU	4	4

Table 4: Execution Latency for Different Instruction Types

Description	**Latency (cycles)**
L1 Search Latency	2
L2 Search Latency	12
L3 Search Latency	60
Main Memory Search Latency	150
Address Generation	1

Table 5: Latencies based on Memory Access Type

Category	**Includes**	**Likely to include**
1 cycles	ALU Instructions, Branches, Stores	
2 – 11 cycles	FP, Slow ALU	Loads that hit in L1
12 – 59 cycles		Loads that hit in L2
60 – 149 cycles		Loads that hit in L3
150+ cycles		Loads that miss in L3

Table 6: Classification by Execution Latency

From Figure 32, we see that around 50% of the instructions are single cycle instructions whereas 43% of them take anywhere between 2 to 11 cycles. Less than 7% of the instructions take 12 or more cycles. While Figure 34 indicates that the benefit per correct prediction is very high for instructions that take 12 or more cycles, the overall speedup for them in Figure 33 is low owing to poor coverage. Instructions in the 2-11 cycles category provide the highest overall speedup, followed by instructions in the 1 cycle category. In summary, it may not be feasible to design a predictor that targets only instructions that miss in the L1 cache simply because there are very few of them. However, it may be beneficial to predict values for them even when the prediction confidence is low as the benefit per correct prediction is really high. Another inference is that single cycle instructions are worth predicting since they bring majority of the coverage.

Note that this experiment is not perfect as instruction latencies are noted from execution runs with value prediction disabled. On enabling value prediction, the latencies of some instructions may change.

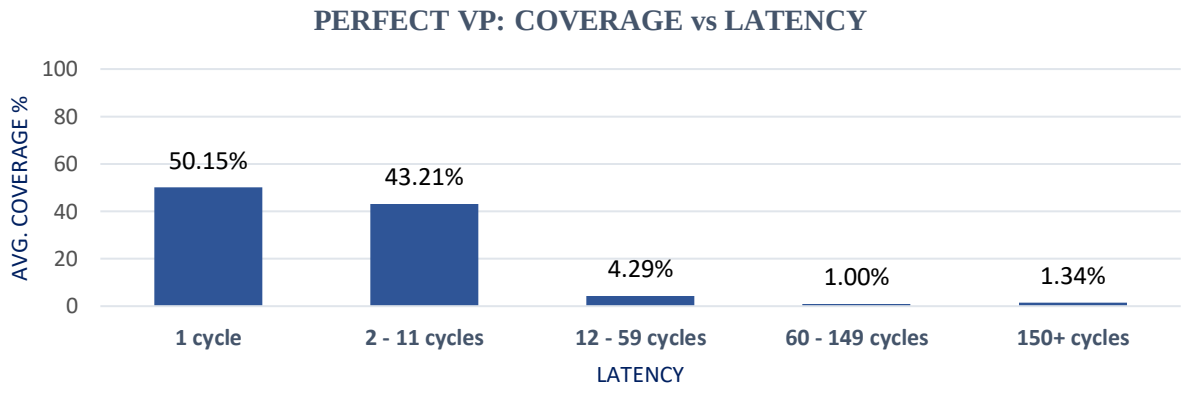

Figure 32: Classification by Instruction Latency: Coverage

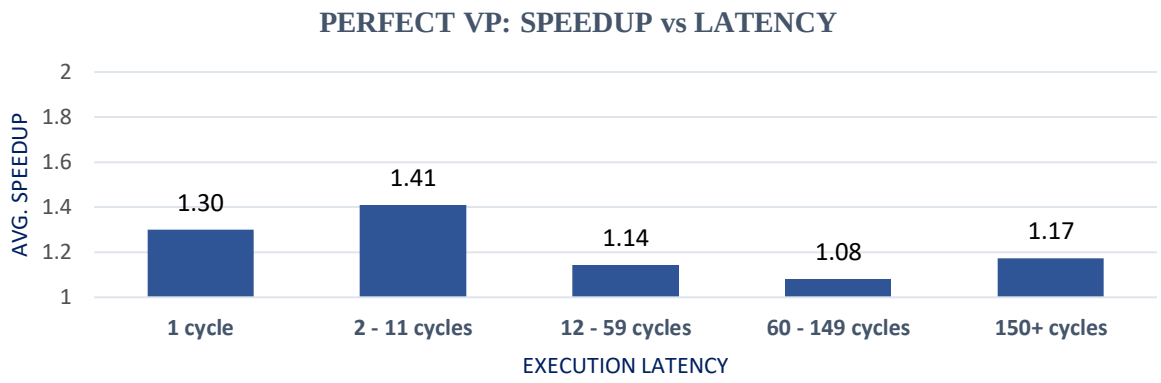

Figure 33: Classification by Instruction Latency: Speedup

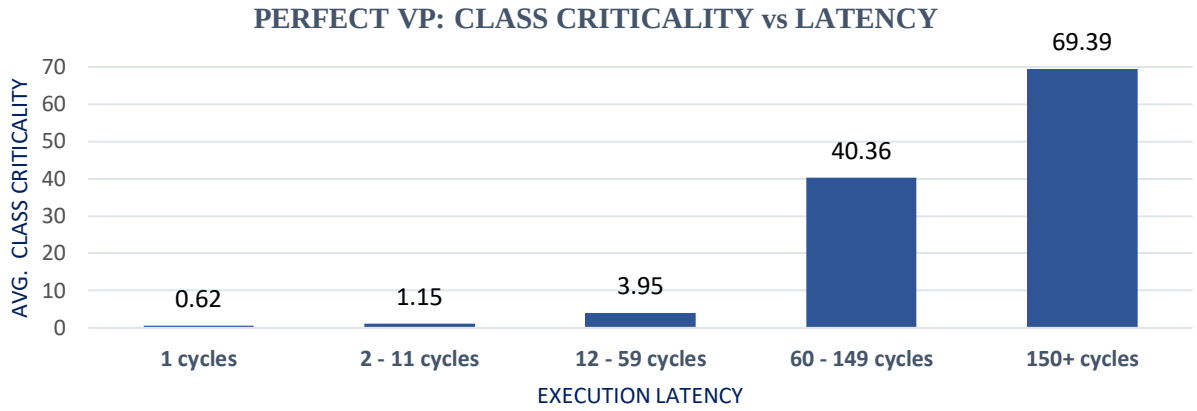

Figure 34: Classification by Instruction Latency: Class Criticality

4.2.2 Classification by Instruction Type (opcode)

We classify instructions by their opcodes and perform perfect value prediction on each one of the instruction types. Figure 35 tells us that ALU instructions and loads constitute almost 88% of the total number of prediction-eligible instructions. From Figure 36, we see that loads alone can provide 89% speedup which is far greater than those provided by other instruction types. This is not surprising as loads have high coverage and will at least have twice the latency compared to an ALU instruction. There has been prior work on predictor designs that target load instructions exclusively [4]. Per prediction benefit is quite low for ALU instructions. However, the overall speedup is a

significant 24% indicating that their high coverage makes them the second most important instruction type to predict for.

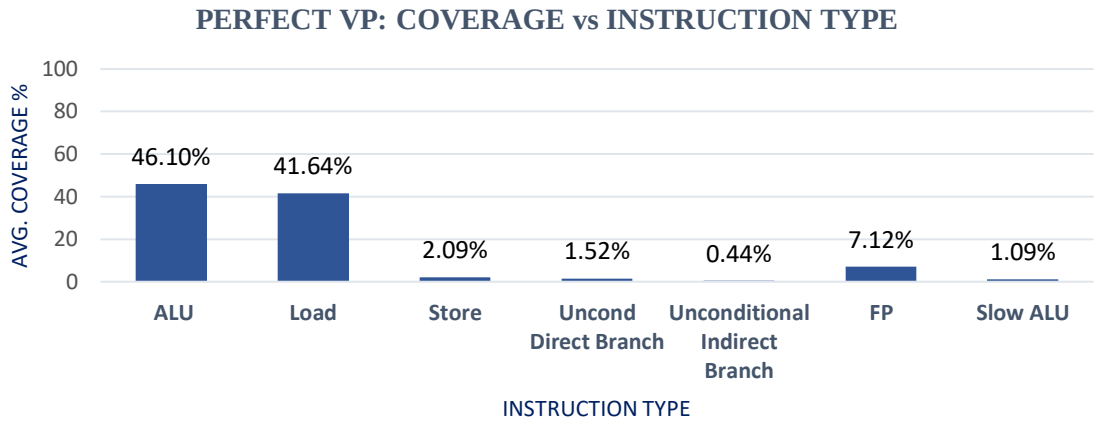

Figure 35: Classification by Instruction Type: Coverage

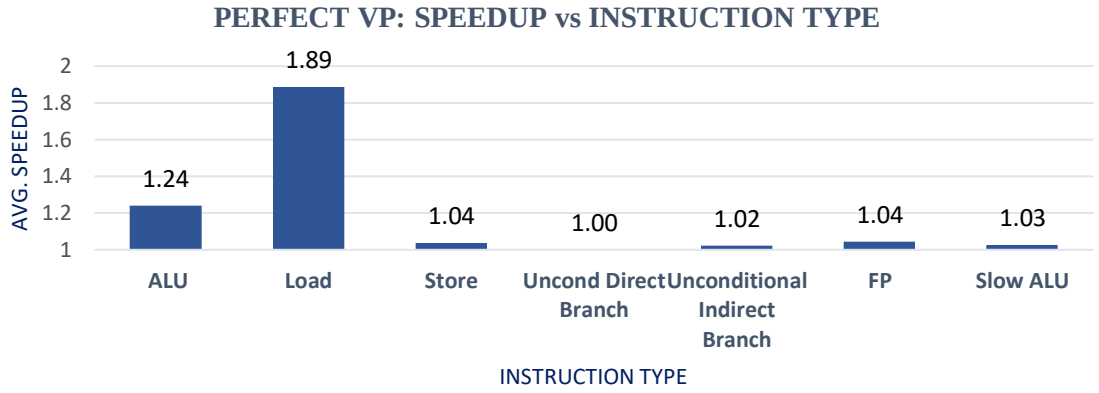

Figure 36: Classification by Instruction Type: Speedup

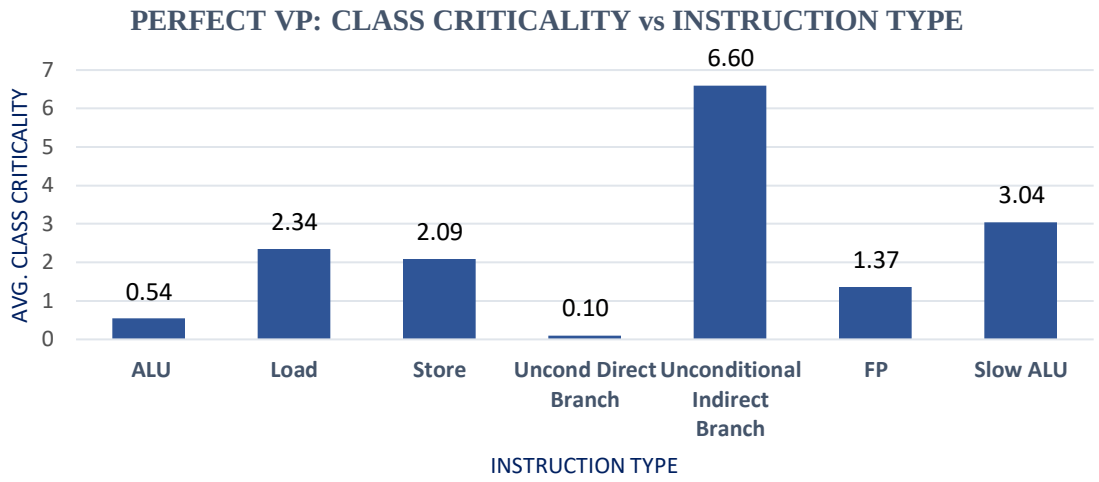

Figure 37: Classification by Instruction Type: Class Criticality

4.2.3 Classification based on whether values are addresses

Some instructions produce addresses as values. We identify such instructions by comparing their values with the data or instruction addresses accessed by the subsequent instructions. If they match, we assume that the value produced was an address for a subsequent memory access. We only look at the subsequent 256 instructions as any instruction beyond that will not lie in the same instruction window and hence will not be fetched before the producer instruction retires. Further, some of the address producing instructions may themselves be loads. It may be beneficial to predict values for these instructions to enable memory level parallelism. In the absence of value prediction, the dependent memory access will have to follow the address producing load, causing serialization of memory accesses. With these possibilities in mind, we classify the instructions as shown in Table 7.

Category	Description
NoAddr	Instructions whose values are not addresses
DataAddr	Instructions whose values are data addresses
InstrAddr	Instructions whose values are instruction addresses
DataAddrLoad	Loads whose values are data addresses
InstrAddrLoad	Loads whose values are instruction addresses
AllAddr	Instructions whose values are either data or instruction addresses
AllAddrLoad	Loads whose values are either data or instruction addresses

Table 7: Classification based on whether instructions produce addresses

From Figure 40, we see that the benefit per prediction is high for address producing instructions compared to other instructions. We also see that the benefit is higher if the address producing instruction is itself a load. Figure 39 tells us that the headroom speedup for address producing instructions in general is around 38% whereas it is about 31% for address producing load instructions. While the headroom speedup for non address producing instructions is higher owing to higher coverage, it is not possible to ignore the headroom for address producing ones. We believe the speedup can be increased further if we used the address to access memory in advance or to perform a prefetch into the cache. However, since we do not have access to the internals of the CVP

simulator, we do not pursue this further. It is also possible that addresses are more predictable owing to well defined or repetitive data structure access patterns. There has been some prior work on predicting addresses instead of values for loads [4].

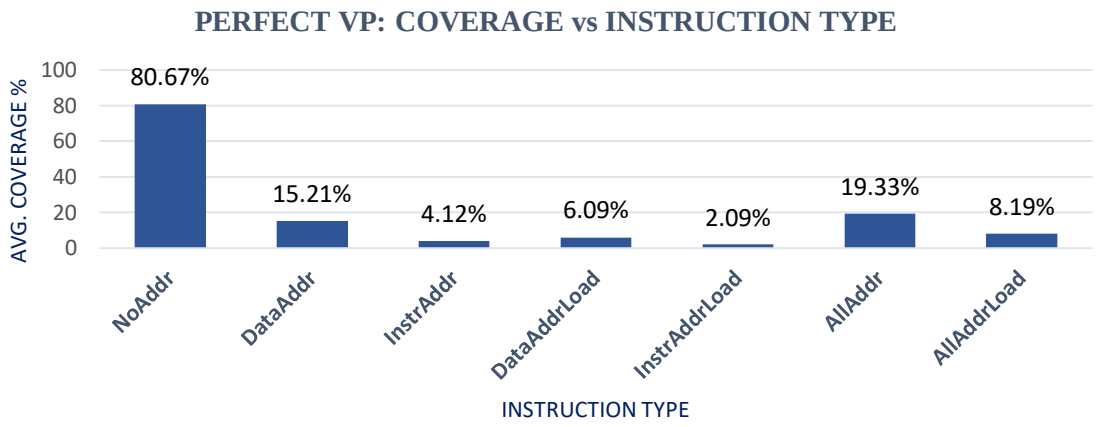

Figure 38: Classification based on whether values are addresses: Coverage

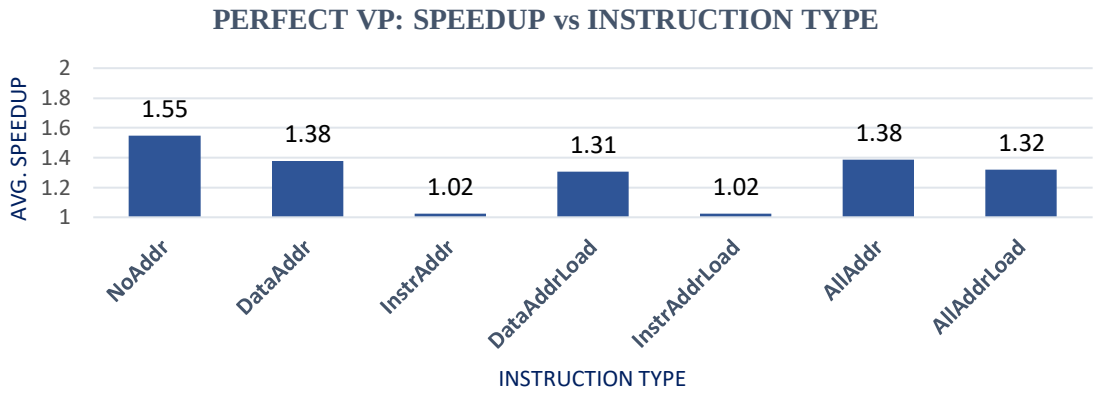

Figure 39: Classification based on whether values are addresses: Speedup

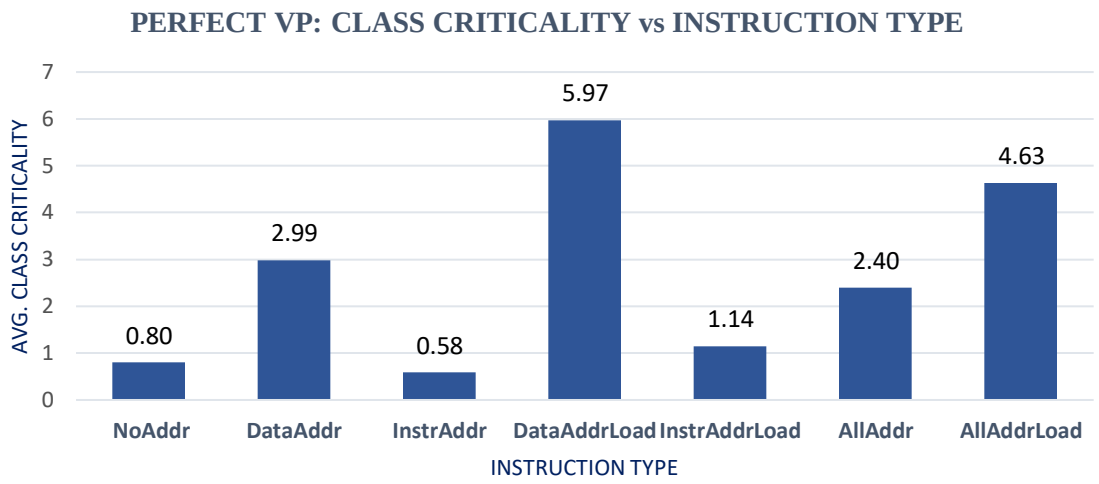

Figure 40: Classification based on whether values are addresses: Class Criticality

4.2.4 Classification based on Instruction Fanout

Some instructions may have more dependent instructions than others. We define fanout as the number of direct dependent instructions that a given value producing instruction has. We speculate that an instruction with high fanout is likely to lie on the critical path of execution due to stalling of a larger number of dependent instructions. We categorize instructions by their fanouts and perform perfect value prediction on them. From Figure 43, we see that the benefit per prediction is significantly higher for instructions with a fanout of 3 or more. Figure 42 suggests that we may be able to achieve good speedup if we predicted selectively for these instructions. As shown in Figure 41, less than 30% of the instructions have a fanout of 2 or more and less than 15% of the instructions have a fanout of 3 or more. This indicates that it may be possible to track high fanout instructions alone using less predictor state. A functional predictor will either require static knowledge of which instructions have high fanout or will have to identify them dynamically.

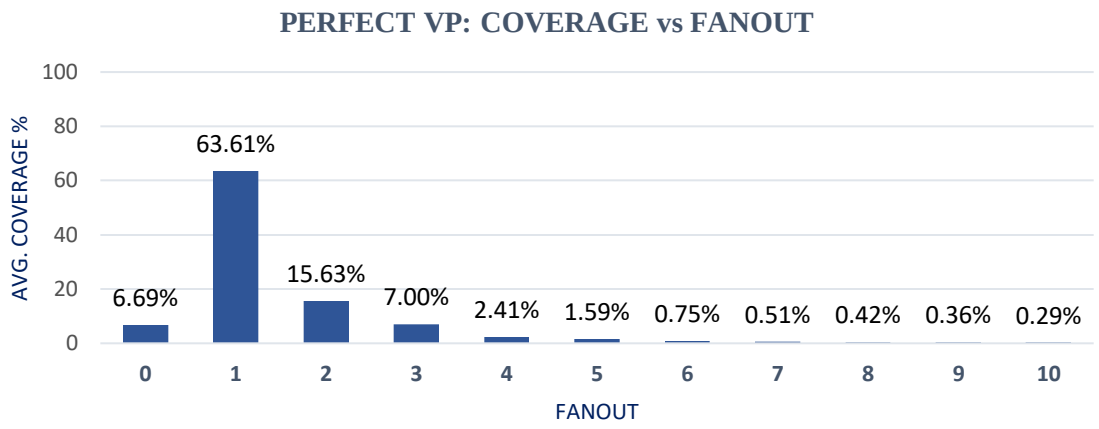

Figure 41: Classification based on instruction fanout: Coverage

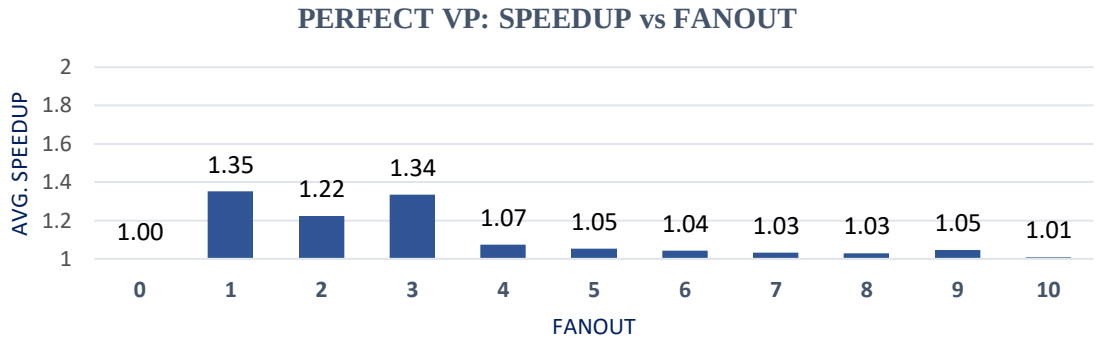

Figure 42: Classification based on instruction fanout: Speedup

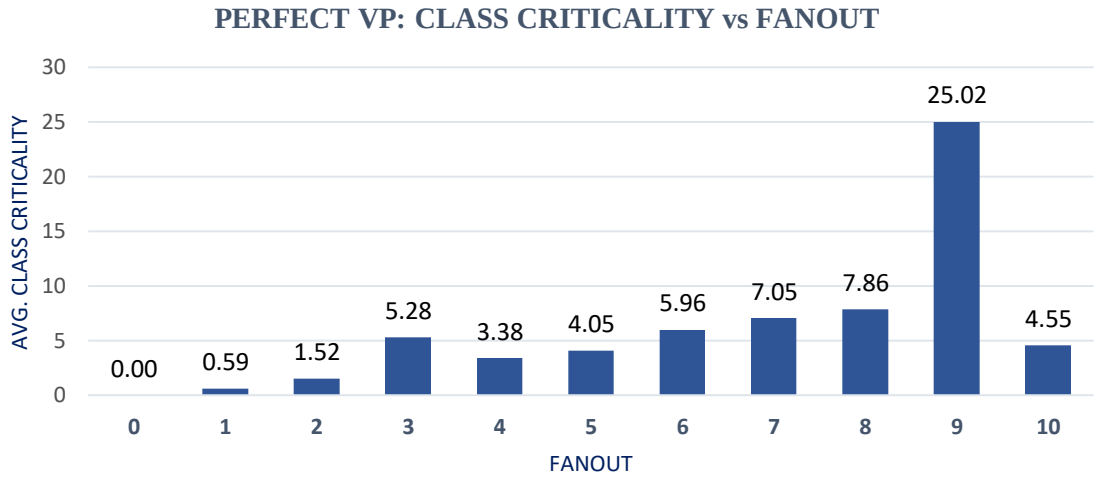

Figure 43: Classification based on instruction fanout: Class Criticality

4.2.5 Classification based on Distance to Nearest Dependent Instruction

Distance to nearest dependent instruction is the number of instructions that are fetched after the value producing instruction until its first consumer instruction. We speculate that the benefit per correct prediction may be higher if the consumer instruction is closer to the producer instruction as there may not be enough cycles before dispatch of the dependent instruction to hide the execution latency of the producer instruction. We classify instructions based on the distance to the nearest dependent instruction and perform perfect value prediction on them. Figure 44 tells us that around 41% of the instructions are followed immediately by a dependent instruction. The coverage reduces for instructions with longer distances to the nearest dependent instruction. We see the effects of coverage reflect on the overall speedup in Figure 45. However, we do not see any trends in benefit per prediction in Figure 46. We assume this may be because of the superscalar design where a bunch of instructions are fetched and dispatched at once, causing many dependent instructions to stall at roughly the same time irrespective of distance to the producer instruction.

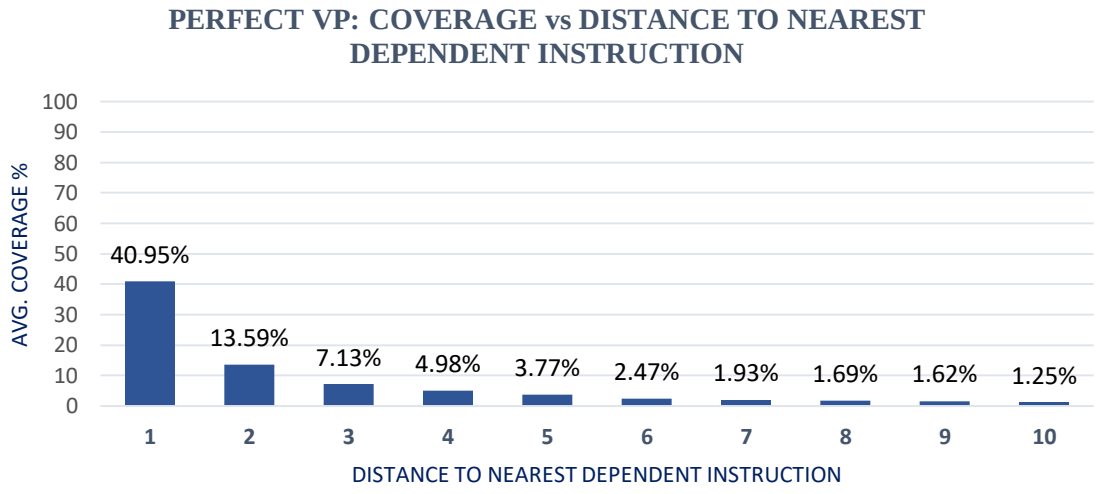

Figure 44: Classification based on distance to nearest dependent instruction: Coverage

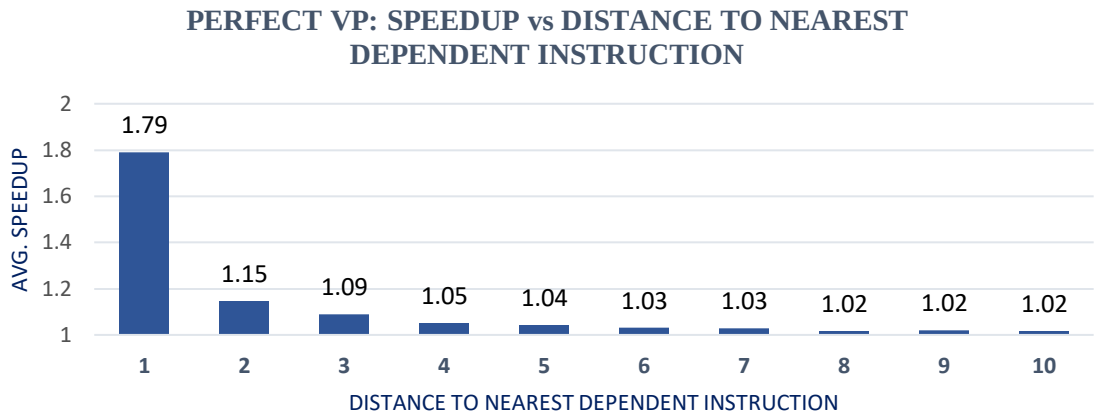

Figure 45: Classification based on distance to nearest dependent instruction: Speedup

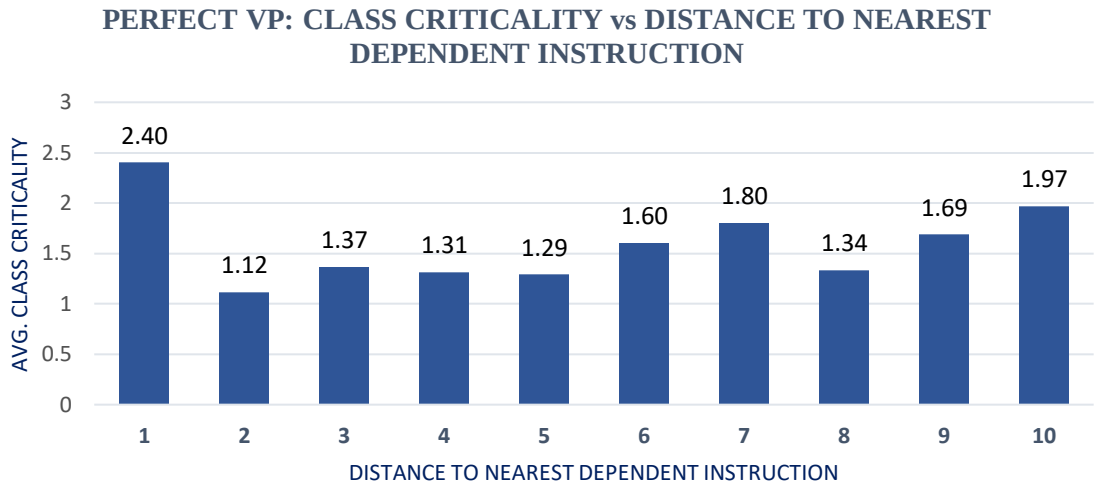

Figure 46: Classification based on distance to nearest dependent instruction: Class Criticality

4.3 Conclusions and Future Work

To conclude, we made an attempt at categorizing instructions in different ways to identify classes that can be specifically targeted or prioritized for value prediction. We believe that value prediction on load instructions, address producing instructions and high fanout instructions can be extremely beneficial. While some of these ideas have been known before, we believe that our work quantifies headroom and illustrates tradeoffs in

coverage and benefit per correct prediction. Until now there has been no prior work in value prediction that targets high fanout instructions specifically. We believe this is worth investigating in future.

References

[1] Arthur Perais. Increasing the performance of superscalar processors through value prediction. Hardware Architecture [cs.AR]. Universit Rennes 1, 2015. English.

[2] A. V. Nori, J. Gaur, S. Rai, S. Subramoney and H. Wang, "Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-Level Cache Hierarchies," *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, 2018, pp. 96-109.

[3] A. Seznec, "Exploring value prediction with the eves predictor," in First Championship Value Prediction, CVP 2018, Los Angeles, June 3, 2018.

[4] R. Sheikh, H. W. Cain, and R. Damodaran, "Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores," in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-50 '17, 2017.

[5] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, 2014, pp. 428-439.

[6] A. Perais and A. Seznec, "EOLE: Paving the way for an effective implementation of value prediction," *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, 2014, pp. 481-492.

[7] “First championship value prediction.” in https://www.microarch.org/cvp1, 2018.

[8] N. Deshmukh, S. Verma, P. Agrawal B. Panda M Chaudhuri, “DFCM++: Augmenting DFCM with Early Update and Data Dependency-driven Value Estimation,” *First Championship Value Prediction, CVP 2018*, Los Angeles, June 3, 2018.

[9] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” *SIGPLAN Not.*, vol. 31, Sep. 1996.

[10] F. Gabbay, “Speculative execution based on value prediction,” EE Department TR 1080, Technion - Israel Institue of Technology, Tech. Rep., 1996.

[11] A. Seznec, “A new case for the tage branch predictor,” in *44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. ACM, 2011.

[12] A. Seznec, “A 64-kbytes ittage indirect branch predictor,” in *Third Championship Branch Prediction*, ser. JWAC-2, 2011.

[13] G.M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities.” *Spring Joint Computer Conference*, pages 483485, 1967.

[14] Y. Sazeides and J. E. Smith. “The predictability of data values.” *The 30th Annual International Symposium on Microarchitecture*, Dec. 1997

[15] B. Goeman, H. Vandierendonck, K. Bosschere, ”Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency”, *Seventh Int’l Symp. High Performance Computer Architecture*, pp. 207-216, 2001-Jan.

[16] B. Fields, S. Rubin, and R. Bod´ık. “Focusing processor policies via critical-path prediction.” *International Symposium on Computer Architecture*, pages 74–85, 2001.

[17] P. Joshi “Techniques to Advance Value Prediction” Master’s Thesis, The University of Texas at Austin, 2019.

[18] M. H. Lipasti and J. P. Shen. “Exceeding the dataflow limit via value prediction.” *MICRO*, pages 226–237. IEEE Computer Society, 1996.

[19] Y. Sazeides and J. Smith. “Implementations of context based value predictors. “ Technical report, University of Wisconsin Madison, 1998.

[20] R. Thomas and M. Franklin. “Using dataflow based context for accurate value prediction.” *PACT*, pages 107–117, 2001.

[21] R. J. Eickemeyer and S. Vassiliadis. “A load-instruction unit for pipelined processors.” *IBM Journal of Research and Development*, 37(4):547–564, Jul. 1993.

[22] H. Zhou, J. Flanagan, and T. M. Conte. “Detecting global stride locality value streams.” *International Symposium on Computer Architecture*, pages 324–335, 2003.

Vita

Anjana Subramanian was born in Bangalore, India. She received her Bachelor of Engineering degree in Telecommunication Engineering from M S Ramaiah Institute of Technology in June 2015. She joined The University of Texas at Austin in August 2017 to pursue a Master's degree in Electrical and Computer Engineering. Between 2015 and 2017, she worked for the Architecture and Technology Group at Arm, Bangalore. She did an internship at Apple Inc. during the summer of 2018.

Permanent email ID: anjana2693@gmail.com

This dissertation was typed by the author.